

---

# Estudio de clases de complejidad de testing

---

Universidad Complutense de Madrid  
Facultad de Informática



Cristhian Rodríguez (Ing. Software)  
Jonathan Carrero (Ing. Software)  
Yu Liu (Ing. Informática)

*Dirigido por:* Ismael Rodríguez Laguna

Curso 2017/18



*“Un hombre provisto de papel, lápiz y goma, y con sujeción a una disciplina estricta,  
es en efecto una máquina de Turing universal.”*

Alan Mathison Turing

## **Agradecimientos**

**Cristhian Rodríguez,**

Quiero agradecer a Ismael por brindarnos la oportunidad de realizar este proyecto y habernos guiado durante el transcurso de este. Agradezco también a mis compañeros del proyecto, Jonathan y Yu, por hacer posible que el proyecto salga adelante. A todos mis compañeros y profesores de clases, más concretamente a Miguel y Sergio, quienes han estado ahí para apoyarme en todo momento. Finalmente, agradezco de todo corazón a mi madre, una mujer muy luchadora, que ha hecho posible que yo pueda hacer esta carrera. Y a mi hermana, por ser en todo momento un referente para mí.

**Jonathan Carrero,**

En primer lugar, quisiera dar gracias a las personas sin las que ahora mismo ni siquiera podría estar dando las gracias; a mis compañeros del proyecto, Cristhian y Yu, y a nuestro director, Ismael. También quiero dar gracias a todos aquellos familiares, amigos y profesores que me dieron ánimos durante esta etapa que finaliza; habéis sido las alas de este vuelo. Por último, agradezco a todas las personas que, independientemente del campo, ponen a disposición de los demás todo su conocimiento, haciendo del mundo un lugar mejor.

A todos, gracias.

**Yu Liu,**

Primeramente, quisiera dar las gracias a mis padres, que siempre me apoyaron incondicionalmente en la parte moral y económica para poder seguir mis estudios en España. Asimismo, quiero agradecer a la Universidad Complutense de Madrid por haberme aceptado y ser parte de ella. Mi agradecimiento también va dirigido a mis compañeros del proyecto, Jonathan y Cristhian, y a nuestro director, Ismael. Finalmente, agradezco a todos los compañeros de clase y profesores que me ayudaron en esta etapa que finaliza.

## Índice de contenido

Agradecimientos .....	4
Resumen / Abstract .....	11
Palabras clave / Keywords.....	12
Listado de abreviaturas por abecedario.....	13
1. Introducción .....	14
1.1. Motivación del proyecto .....	14
1.2. Objetivos .....	14
1.2.1. Objetivos durante la Fase I.....	14
1.2.2. Objetivos durante la Fase II.....	15
1.3. Plan de trabajo .....	15
1.3.1. Tareas generales a realizar .....	16
1.3.2. Reuniones de seguimiento.....	16
1.4. Estructura de la memoria .....	17
2. Trabajo relacionado .....	18
2.1. Aproximación a los artículos de investigación .....	18
2.1.1. A General Testability Theory .....	18
2.1.2. Introducing Complexity to Testing.....	20
2.2. Mutation Testing.....	23
2.2.1. Finalidad del mutation testing .....	23
2.2.2. Visión general.....	24
2.3. Generación de mutantes .....	25
2.3.1. Pitest.....	26
3. Desarrollo del proyecto .....	29
3.1. Ideas iniciales descartadas .....	29
3.2. Comprendiendo el modelo a desarrollar .....	31
3.2.1. Introducción a la primera fase del proyecto .....	33
3.2.2. Introducción a la segunda fase del proyecto.....	33
3.3. Fase I: mutaciones en árboles binarios.....	35
3.3.1. Implementación y funcionamiento.....	36
3.3.2. Los tres escenarios a testear .....	39
3.4. Fase II: mutaciones en Programas reales y Programas generados .....	44

3.4.1.	Visión general.....	44
3.4.2.	Interfaz de usuario.....	45
3.4.3.	Crear proyecto con Programa real.....	47
3.4.4.	Crear proyecto con programa generado.....	50
3.4.5.	Preprocesado .....	57
3.4.6.	Base de datos .....	57
4.	Análisis de experimentos .....	59
4.1.	Análisis de la Fase I.....	59
4.1.1.	Análisis de los dos primeros ejemplos simulados .....	60
4.1.2.	Análisis del tercer ejemplo simulado .....	62
4.2.	Análisis de la Fase II .....	63
4.2.1.	Análisis de programas reales.....	64
4.2.2.	Análisis de programas generados.....	66
4.2.3.	Análisis del impacto en los tipos de mutaciones.....	70
5.	Conclusiones / Conclusions.....	73
6.	Reparto de tareas.....	77
6.1.	Cristhian Rodríguez.....	77
6.2.	Yu Liu .....	78
6.3.	Jonathan Carrero.....	80
7.	Repositorios <i>Github</i> .....	82
8.	Tecnologías utilizadas .....	83
8.1.	Herramientas de desarrollo .....	83
8.1.1.	Editores .....	83
8.1.2.	Software .....	83
8.2.	Herramientas de gestión .....	85
8.2.1.	Control de versiones .....	85
8.2.2.	Organización del proyecto .....	86
9.	Referencias .....	98

## Índice de Ilustraciones

Ilustración 1: Diagrama de Venn que representa las clases .....	19
Ilustración 2: Distinción entre pares de funciones .....	20
Ilustración 3: Secuencias de conjuntos finitos.....	21
Ilustración 4: Generando mutantes.....	24
Ilustración 5: Aplicando el Test Suite .....	25
Ilustración 6: Informe de cobertura de Pitest .....	27
Ilustración 7: Modelo de la primera idea .....	29
Ilustración 8: Representación teórica de los distintos árboles.....	30
Ilustración 9: Representación real de los distintos árboles.....	30
Ilustración 10: Consecuencia al matar un mutante .....	31
Ilustración 11: Una sola definición correcta de la IUT .....	32
Ilustración 12: Escenario real de definiciones de la IUT .....	33
Ilustración 13: Modelo introductorio a la fase II .....	34
Ilustración 14: Representación de un árbol binario .....	35
Ilustración 15: Representación en array de un árbol binario por niveles .....	35
Ilustración 16: Aplicando test suite .....	38
Ilustración 17: Aplicando un test .....	39
Ilustración 18: DR primer ejemplo aplicando test suite hasta nivel 4 .....	40
Ilustración 19: DR segundo ejemplo aplicando test suite hasta nivel 4 .....	41
Ilustración 20: DR tercer ejemplo aplicando test suite hasta nivel 6.....	42
Ilustración 21: Mutante del segundo ejemplo .....	43
Ilustración 22: Estructura del proyecto <i>GUI-master</i> .....	44
Ilustración 23: Ejecución natural del proyecto .....	45
Ilustración 24: Vista de la opción Resultados con dos proyectos.....	46
Ilustración 25: Tipos de mutaciones a aplicar sobre proyectos Java .....	46
Ilustración 26: DR <i>Programa personajes</i> aplicando diez tests.....	49
Ilustración 27: DR <i>Programa math</i> aplicando diez tests .....	50
Ilustración 28: Esquema de la base de datos .....	58
Ilustración 29: CT en primer ejemplo con test suite.....	60
Ilustración 30: CT en segundo ejemplo con test suite .....	60
Ilustración 31: Función descrita por $\log(n) \cdot \log(\log(n))$ .....	61
Ilustración 32: CT en tercer ejemplo con test suite.....	62
Ilustración 33: Función descrita por $O(n \cdot \log(n))$ .....	63
Ilustración 34: CT en <i>Programa personajes</i> .....	65
Ilustración 35: CT en <i>Programa math</i> .....	65
Ilustración 36: DR obtenido incrementando un solo atributo.....	68
Ilustración 37: DR obtenido incrementando dos atributos .....	70
Ilustración 38: DR según el tipo de mutación .....	71
Ilustración 39: Cobertura de código en paquete <i>Math</i> .....	91
Ilustración 40: Cobertura de código en paquete <i>figura2D</i> .....	91

Ilustración 41: Cobertura de código en paquete <i>figura3D</i> .....	91
Ilustración 42: Cobertura de código en paquete <i>personaje</i> .....	92



## Índice de Tablas

Tabla 1: Varianza del DR en programas con parámetros de generación diferentes.....	64
Tabla 2: Tipos de mutaciones en Pitest .....	88
Tabla 3: Método principales de <i>ArrayList</i> .....	89
Tabla 4: Comando Unix utilizados.....	96

## Índice de Anexos

<b>Anexo I:</b> Hipótesis en las pruebas.....	87
<b>Anexo II:</b> Tipos de mutaciones en Pitest .....	87
<b>Anexo III:</b> Características de Java .....	88
<b>Anexo IV:</b> Métodos principales usados en la estructura de <i>ArrayList</i> .....	88
<b>Anexo V:</b> Efecto frontera .....	89
<b>Anexo VI:</b> Estructura de carpetas.....	89
<b>Anexo VII:</b> Cobertura de código.....	90
<b>Anexo VIII:</b> Parámetros del <i>Programa generador</i> .....	92
<b>Anexo IX:</b> Gramática Incontextual.....	93
<b>Anexo X:</b> Secuencia de tareas del fichero <i>preprocesar.sh</i> .....	93
<b>Anexo XI:</b> Comandos Unix .....	95
<b>Anexo XII:</b> Características de la base de datos.....	96

## Resumen / Abstract

La importancia que está tomando la tecnología en nuestros días obliga a las empresas a invertir grandes cantidades de recursos para asegurar que sus sistemas informáticos funcionan con la mayor fiabilidad posible. Gran parte de estos recursos son utilizados en las pruebas iniciales que se llevan a cabo a la hora de testear los sistemas. En este trabajo estudiaremos la complejidad que supone aplicar dichas pruebas, identificar qué sistemas, según su naturaleza, suponen una mayor complejidad al ser testeados o en qué momento no merece la pena continuar destinando recursos para tratar de mejorar la confianza de los mismos. Para ello, desarrollaremos herramientas que nos permitan realizar experimentos en diferentes escenarios de prueba, analizar resultados y razonar sobre las conclusiones obtenidas.

---

The importance of technology today is forcing companies to invest large amounts of resources to ensure that their IT systems operate as reliably as possible. A large part of these resources are used in the initial tests that are carried out at the time of testing the systems. In this work we will study the complexity of applying these tests, identify which systems, according to their nature, are more complex when they are tested or when it is not worthwhile to continue allocating resources to try to improve their confidence. To this end, we will develop tools that allow us to conduct experiments in different test scenarios, analyze results and reason on the conclusions obtained.

## Palabras clave / Keywords

Testing, pruebas de mutación, complejidad de testing, servicio web, árboles binarios, estructuras de datos.

---

Testing, mutation testing, testing complexity, web services, binary trees, data structures.

## Listado de abreviaturas por abecedario

- CT: Complejidad de Testing
- DR: Distinguishing Rate
- HTTP: Hypertext Transfer Protocol
- IDE: Integrated Development Environment
- IUT: Implementation Under Test
- JVM: Java Virtual Machine
- PIT: Pitest
- POM: Project Object Model
- RAM: Random Access Memory
- TFG: Trabajo de Fin de Grado
- XAMPP: X (cualquier sistema operativo), Apache, MariaDB/MySQL, PHP, Perl

## 1. Introducción

### 1.1. Motivación del proyecto

Durante gran parte de la carrera ha habido momentos que nos han obligado a dar prácticamente el 100% de nosotros mismos, pero al final, la saturación que hayamos podido tener en la mayoría de los casos era producida por el número y la *anchura* que cobraban los trabajos que debíamos hacer y no por la *profundidad* en los mismos.

Para este Trabajo de Fin de Grado queríamos algo diferente. Algo que tuviese un objetivo claro y que llegar a él supusiese un reto. Queríamos realizar un proyecto que nos obligase a pensar constantemente antes de actuar.

### 1.2. Objetivos

El objetivo principal del proyecto es corroborar que los resultados teóricos obtenidos en el artículo *Introducing Complexity to Testing* [1] (basado a su vez en el artículo previo *A General Testability Theory: Classes, Properties, Complexity, and Testing Reductions* [2]), acerca de cómo aumenta el esfuerzo que tenemos que dedicar a aplicar pruebas para aumentar nuestro nivel de confianza en la corrección del sistema (la llamada complejidad de testing) se mantiene también cuando dichas pruebas son simuladas empíricamente. Además, yendo un paso más allá, aplicaremos pruebas a programas reales para tratar de averiguar qué características de los mismos (relativas a su estructura o a los errores que el programador tiende a cometer) afectan más al esfuerzo necesario para detectar potenciales errores al realizar pruebas. Es por eso que dividiremos los objetivos del proyecto en dos fases.

#### 1.2.1. Objetivos durante la Fase I

- Comprobar empíricamente si la complejidad de testing (de ahora en adelante denotada simplemente como CT) prevista en dicho artículo es la misma cuando las pruebas se simulan empíricamente. Estas pruebas, como veremos en breve, estarán basadas en aplicar baterías de tests y medir cómo aumenta la fiabilidad de los programas a medida que se aplican.
- Desarrollar un programa en Java que permita aplicar pruebas de testing en sistemas representados como árboles de ejecución<sup>1</sup> que se corresponden con los ejemplos teóricos del artículo.
- Analizar y sacar conclusiones sobre los resultados obtenidos basándonos en la información que aporten los experimentos realizados y poder contrastarla con los resultados teóricos del artículo.

---

<sup>1</sup> Un árbol de ejecución (más adelante veremos que concretamente se trata de árboles binarios) es una estructura de datos en la cual cada nodo puede tener un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos (de ahí el nombre "binario"). Si algún hijo tiene como referencia a *null*, es decir que no almacena ningún dato, entonces este es llamado un nodo externo.

### 1.2.2. Objetivos durante la Fase II

- Realizar experimentos para detectar qué características de los programas tienen un mayor impacto en la CT al aplicar pruebas. Concretamente, se trata de identificar cómo varía la CT cuando aplicamos las pruebas sobre programas con distintas estructuras (e.g. variación en el número de condiciones de bifurcación, nivel de anidamiento, iteración en los bucles, etc.) y diferentes errores producidos en ellas.
- Desarrollar un servicio web que disponga de una base de datos desde la que exportar/importar resultados que permitan construir gráficas de complejidad cuando las pruebas de testing son aplicadas a proyectos Java.
- Desarrollar programas lo más realistas posibles (e.d. programas parecidos a lo que un programador podría enfrentarse) que nos permitan observar CT variadas al aplicar pruebas sobre ellos.
- Desarrollar un programa en Java que, de manera automática, permita generar otros programas Java para utilizarlos en nuestros experimentos. Además, dicho programa debe ser capaz de recibir ciertos parámetros que indiquen la estructura de los programas que serán generados (número de condiciones de bifurcación, nivel de anidamiento, número de iteraciones en los bucles, etc).
- Integrar un método para aplicar pruebas de testing. En este aspecto, es necesario que seamos capaces de, tomando como base un determinado programa *original* (más adelante detallaremos este concepto), podamos generar muchas copias del mismo en las cuales, de alguna manera, introduzcamos fallos<sup>2</sup> como los que podría tener un programador a la hora de desarrollar un programa a mano.
- Analizar y sacar conclusiones sobre los resultados obtenidos basándonos en las distintas pruebas aplicadas a los diferentes escenarios que hayamos desarrollado.

### 1.3. Plan de trabajo

Debido a que intuimos la dificultad que conllevaría entender los artículos de investigación, planificamos dedicar el primer mes del proyecto para la lectura y comprensión de los mismos, teniendo reuniones semanales con nuestro director. Una vez que entendimos y establecimos los objetivos del TFG, dedicamos unas dos semanas más para pensar cómo desarrollar una aplicación que simulase con la mayor fidelidad posible los ejemplos allí estudiados. Desarrollo que, según se estimó, ocuparía prácticamente el resto del primer cuatrimestre (hasta mediados de enero) incluidas las pruebas, análisis y conclusiones de esta primera fase.

En cuanto al desarrollo de la segunda fase del proyecto, se llevaría a cabo durante los siguientes cuatro meses (desde enero hasta abril, ambos incluidos). Esta parte del proyecto tendría una mayor carga de trabajo en cuanto a la programación de código, pues se

---

<sup>2</sup> Cuando hablamos de los fallos que puede tener un programador es importante destacar que no nos referimos a fallos que supongan errores de compilación en los programas, sino más bien a fallos cuya implicación afecta al resultado final de la ejecución.

desarrollaría un servicio web para facilitar la interacción con el usuario, así como un programa que sea capaz de generar otros programas de manera automática.

Por último, prevemos dedicar el mes de mayo a posibles correcciones sobre la memoria y comprobar si se cumplieron los objetivos que se establecieron en un principio.

### **1.3.1. Tareas generales a realizar**

En nuestros primeros experimentos analizaremos la CT en diferentes escenarios de prueba sacados del artículo *Introducing Complexity to Testing*. Por ello, la tarea principal durante la primera fase consistirá en crear una aplicación que sea capaz de simular empíricamente dichos escenarios.

Ya en la segunda fase, pretendemos desarrollar una aplicación que no sólo que funcionalmente cumpla con los requisitos que establezcamos, sino que también sea una herramienta para aquellas personas que deseen aplicar baterías de pruebas sobre proyectos que ellos mismos hayan hecho, con la finalidad de medir la CT. En tal sentido, nuestra idea es la de desarrollar un servicio web donde podamos subir proyectos, realizar pruebas sobre ellos y obtener, a través de una interfaz agradable para el usuario y extrayendo los datos de una base de datos, gráficas de las cuales inferir resultados. Al mismo tiempo que estamos interesados en estudiar la CT de programas hechos de forma manual, también lo estamos en desarrollar un programa que genere automáticamente otros programas (a tomar como correctos por definición en cada caso), para poder estudiar sistemáticamente cómo la CT depende de la estructura de los programas generados y de los tipos de mutaciones hechas. De este modo, no tendríamos que programar manualmente diferentes situaciones para forzar que las pruebas a aplicar sean más o menos eficientes en la detección de errores. En tal sentido, dicho programa “generador de programas” recibiría parámetros para saber cómo debe ser la estructura de los programas que genere.

Terminando con esta segunda fase, pretendemos que todo sea integrable entre sí y que un usuario, interaccionando sólo a través de la interfaz del servicio web, pueda subir proyectos hechos por sí mismo/a, generar programas y, además, aplicar baterías de pruebas sobre cualquiera de ellos para visualizar gráficas con la información obtenida.

### **1.3.2. Reuniones de seguimiento**

A medida que nos abstraíamos (hasta cierto punto) del artículo y concretábamos qué es lo que perseguíamos, establecimos que, o bien las reuniones se hicieran cuando nos encontrásemos en un punto de cierta importancia, o cada dos semanas aproximadamente. Durante cada reunión nos marcábamos pequeños objetivos que queríamos conseguir para las siguientes dos semanas, dividiendo así el desarrollo en pequeños hitos que repartíamos entre los tres componentes del equipo. Cuando se finalizaba esa parte del trabajo, se volvía a contactar con el director para corregir o corroborar si el camino por el que íbamos era el



correcto, al mismo tiempo que se explicaban los conceptos que fuesen necesarios para la siguiente reunión.

#### 1.4. Estructura de la memoria

- En primer lugar, nos encontraremos con una parte de *Introducción*. Aquí se hablará de las motivaciones del proyecto, objetivos y del plan de trabajo que se ha llevado a cabo.
- En el segundo punto nos encontraremos con *Trabajo relacionado*. Profundizaremos en todo el trabajo que se ha llevado a cabo sobre el estudio de artículos de investigación, herramientas de software utilizadas y se explicarán conceptos que nos ayudarán a entender el posterior desarrollo.
- Pasamos al tercer punto en el que abarcaremos todo el *Desarrollo del proyecto*. Veremos cómo este TFG está dividido en dos fases y explicaremos en profundidad las etapas de desarrollo que ha tenido cada una de ellas.
- Para finalizar, encontraremos una sección dedicada a los *Análisis de experimentos* y a las *Conclusiones* obtenidas tras realizar las pruebas deseadas.

## 2. Trabajo relacionado

### 2.1. Aproximación a los artículos de investigación

A continuación, se detalla el trabajo previo relacionado con los dos artículos de investigación desde los que partimos para realizar parte de este TFG. Para tener una idea de hacia dónde queríamos ir y qué es lo que teníamos que hacer, tuvimos que entender la base sobre la que se asentarían las pruebas realizadas. Dicha base, recogida en los artículos *A General Testability Theory: Classes, Properties, Complexity, and Testing Reductions* y *Introducing Complexity to Testing*, representó un reto a la hora de entender conceptos con un alto contenido en tecnicismos, fórmulas e ideas que, en muchas ocasiones, supusieron una barrera difícil de superar. Tanto es así que, durante las primeras semanas, los esfuerzos no se centraron en desarrollar, sino más bien en descartar ideas que al poco tiempo nos dábamos cuenta de que no perseguían los objetivos que se pretendían alcanzar.

#### 2.1.1. A General Testability Theory

Este primer artículo nos sirvió como base para entender algunos de los conceptos teóricos que aparecerán durante la explicación del segundo artículo. Puesto que los casos de estudio desarrollados en este TFG conciernen sobre todo a la CT (asunto no tratado en este primer artículo), no consideramos necesario profundizar demasiado en él, pero sí que es importante tener claros algunos conceptos.

Las pruebas que se aplican a un determinado sistema tienen como objetivo comprobar la corrección de dicho sistema interactuando con él. Normalmente, el objetivo de esta interacción es el de comprobar si una implementación cumple una propiedad o una especificación. En relación con lo anterior, se introduce un marco general para definir cualquier escenario particular en el que deseemos realizar pruebas. De ahora en adelante, llamaremos IUT (Implementation Under Test) al sistema real cuya corrección quiere comprobar el testeador vía testing. Veamos algunas de las características que se desarrollan durante el artículo:

- (a) Para proporcionar generalidad, no se asume ninguna estructura específica de sistemas (estados, transiciones, líneas de código, etc.). Los comportamientos están representados por funciones que relacionan las entradas y salidas como si de una caja negra se tratase. Cada posible definición de la IUT (definición completa de su comportamiento) vendrá dada por una función.
- (b) Los diseñadores pueden indicar explícitamente que hay varias definiciones correctas en lugar de una sola, lo cual es un enfoque más natural. De hecho, el objetivo de las pruebas es determinar si la IUT pertenece a un conjunto de definiciones correctas, lo cual es un requisito más débil que identificar unívocamente a la IUT dentro de dicho conjunto.

- (c) Las personas que se encarguen de realizar las pruebas pueden especificar explícitamente qué pares de salidas de las funciones son distinguibles/indistinguibles al ser observados.
- (d) El no determinismo de los sistemas viene denotado explícitamente. Además, el no determinismo afecta a la definición del propio objetivo del testeador: un *test suite* (en español, un *conjunto de pruebas*) se considera completo sólo si, después de ser aplicado a la IUT, siempre se puede decidir si la IUT es correcta o no (independientemente de las elecciones no deterministas hechas por la propia IUT).

En función del tamaño que tengan los tests suites para ser completos, los escenarios de testing se distinguen en varias clases de testeabilidad posibles. Puesto que la explicación en detalle de cada una de ellas queda fuera del alcance de este TFG, tan sólo exponemos una explicación informal de las mismas.

- **Clase I:** en esta clase encontramos escenarios que son testeables de manera finita. Por tanto, existe un conjunto de pruebas finito completo, es decir, que al ser aplicado a la IUT somos capaces de saber si ésta es correcta o no.
- **Clase II:** podemos lograr la completitud de las pruebas en el límite: para cualquier grado de certeza en la corrección de la IUT que deseemos alcanzar, algún test suite finito lo logra. La explicación de esta clase es algo más compleja que el resto y se explicará con mayor detalle en el siguiente artículo.
- **Clase III:** existe un test suite numerable.
- **Clase IV:** existe un test suite completo. Nótese que, a diferencia de la clase anterior, el test suite podría ser no numerable.
- **Clase V:** todos los casos.

Cada clase está estrictamente contenida dentro de la siguiente, como muestra el siguiente diagrama. Téngase en cuenta que la **Clase V** representa todas las posibles definiciones para la IUT, es decir, el conjunto universal.

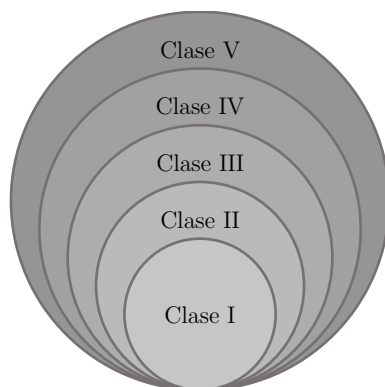


Ilustración 1: Diagrama de Venn que representa las clases

### 2.1.2. Introducing Complexity to Testing

Antes de entrar en definiciones que nos ayuden a entender cómo trataremos la CT a la hora de realizar testing, conviene hacer una mejor definición de la **Clase II** nombrada en el punto anterior. Como dijimos, es quizá la más difícil de comprender e interesante de la jerarquía. Esta clase define una especie de testeabilidad en el límite: a medida que aumenta el tamaño del test suite que se va a aplicar, nuestro conocimiento sobre la corrección o incorrección de la IUT converge a 1, en una escala numérica donde 1 significa completo. De una manera más formal se dice que, dado el conjunto de definiciones que puede tener la IUT (donde algunas de ellas se consideran correctas y otras se consideran incorrectas), la medida que estamos exigiendo que converja a 1 es, de hecho, una tasa de distinción: la proporción por todos los posibles pares<sup>3</sup> que serán distinguidos al observar las salidas que produzca la IUT al aplicar un test suite en cuestión. Cabe destacar que un par se distingue si las salidas producidas nos permiten descartar al menos un elemento del par. En el siguiente ejemplo, el test suite permite distinguir cuatro pares de funciones de los nueve que hay en total. Por ejemplo, supongamos que las funciones  $f_1$ ,  $f_2$  y  $f_3$  son correctas y las funciones  $f_4$ ,  $f_5$  y  $f_6$  incorrectas en la siguiente figura, y que cierto test suite logra distinguir los pares  $\{f_2, f_6\}$ ,  $\{f_3, f_4\}$ ,  $\{f_3, f_5\}$  y  $\{f_3, f_6\}$  marcados en azul.

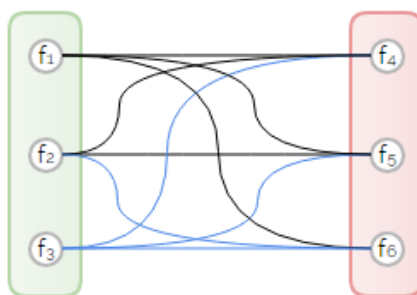


Ilustración 2: Distinción entre pares de funciones

Debe tenerse en cuenta que, si la tasa de distinción es 1, entonces todos los pares se distinguen, por lo que el test suite nos permitiría distinguir la IUT como dentro del conjunto de posibles definiciones correctas o como dentro del conjunto de posibles definiciones incorrectas, aunque no podamos identificarla dentro del conjunto correspondiente. Por tanto, el test suite es completo.

Llegados a este punto, estamos en situación de abordar el concepto de tasa de distinción, de ahora en adelante denominado *Distinguishing Rate* o simplemente DR. Esta tasa relaciona la cantidad de pares que somos capaces de distinguir respecto a los que hay en total. En el ejemplo anterior, la DR obtenida sería  $4/9 = 0.44$  (pues hemos logrado distinguir cuatro pares de los nueve que hay en total). Desgraciadamente, la DR no está definida para conjuntos infinitos, ya que esta tasa es una proporción. Para permitir que la

<sup>3</sup> Cada par es una posible definición correcta del IUT junto con una posible definición incorrecta de esa misma IUT (de ahí que sean pares). Como destacamos en la explicación del artículo anterior, una IUT puede tener un conjunto de definiciones correctas en vez de una única definición correcta.

**Clase II** se ocupe de conjuntos infinitos de posibles definiciones de la IUT, como veremos durante los siguientes párrafos, se considerarán secuencias de conjuntos finitos que convergen con los conjuntos originales, en lugar de conjuntos infinitos en sí mismos. En particular, se dice que un escenario de prueba pertenece a la **Clase II** si, para alguna secuencia como ésta y para todas las DR inferiores a 1, existe algún conjunto finito de pruebas (tests) que permite alcanzar esa tasa de distinción para todos los conjuntos de la secuencia. En la siguiente ilustración podemos ver la idea de considerar secuencias de conjuntos finitos antes nombrada, siendo  $\mathbf{C}$  el conjunto infinito de todas las posibles definiciones que la IUT podría tener.

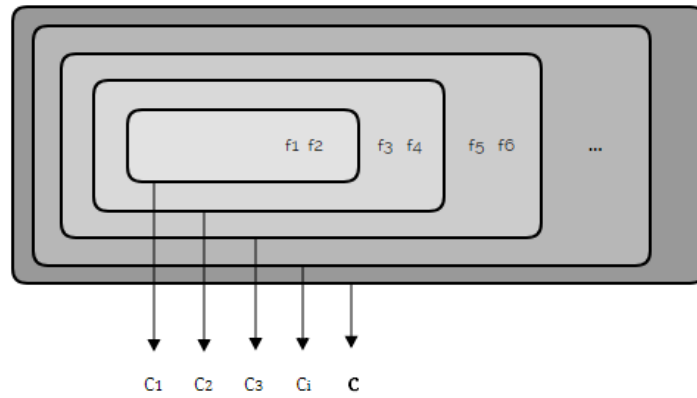


Ilustración 3: Secuencias de conjuntos finitos

Observemos que  $C_1 \cup C_2 \cup C_3 \cup \dots = \mathbf{C}$  y que además se cumple  $C_i \subseteq C_{i+1}$ . Por ejemplo, supongamos que el test suite  $\{i_2, i_3, i_5\}$  alcanza, al menos, un  $DR = 0.7$  para todos los  $C_i$ , que el test suite  $\{i_4, i_7, i_{45}\}$  alcanza, al menos, un  $DR = 0.95$  para todos los  $C_i$ , y que podemos decir algo similar para cualquier  $DR$  objetivo  $< 1$ . Entonces, el escenario de testing pertenece a la **Clase II**. En resumen, si algún test suite finito es capaz de alcanzar cualquier  $DR$ , por muy cercano a 1 que éste sea, para los infinitos  $C_i$ , entonces estaremos en la **Clase II**.

De cara a las pruebas que vamos a realizar durante la primera fase del desarrollo del TFG, nos interesan especialmente tres ejemplos que aparecen en este artículo. Simularemos estos ejemplos mediante una aplicación para comparar si las CT observadas en nuestros experimentos (e.d. la manera en que aumenta el coste de los test suites que debemos aplicar a medida que queremos alcanzar una  $DR$  más cercana a 1) son las mismas que las previstas en los resultados teóricos que aparecen en el artículo.

### Primer ejemplo: un implementador típico que ya realizó algunas pruebas informales

En este primer ejemplo asumiremos las hipótesis IH y MICH (ver **Anexo I**). Este modelo asume que los errores (pasos de ejecución en los que se produce una salida diferente de la esperada) son más probables a medida que la ejecución se hace más larga, es decir, a medida

que vamos profundizando más en el árbol de ejecución de la IUT. Esto supone asumir que un programador competente es menos propenso a cometer errores en los primeros pasos ya que es más probable que sus propias pruebas informales ya hayan verificado el comportamiento de la máquina para iteraciones cortas. Asignando un peso de 1 a cada una de las máquinas y tomando cualquier  $p$  con  $0 \leq p < 1/2$ , a cada máquina posible (función) se le resta peso  $p^k$  por cada salida incorrecta contestada en el paso  $k$ -ésimo de ejecución. Por ejemplo, si  $p = 1/4$ , cada error en el segundo paso restaría  $1/16$ , y cada error en el tercer paso restaría  $1/64$ . Nótese también que, cuando  $p = 0$ , todas las máquinas tienen el mismo peso, es decir, 1.

Asumamos  $p = 1/4$ . La CT es tal que, si se considera que el coste es el número total de *inputs* aplicadas entre todos los tests de un test suite (e.g. un test suite consistente en cuatro tests, donde cada uno aplica consecutivamente dos inputs, cuesta 8), y se asume que los test suites consisten en aplicar todas las secuencias posibles de *inputs* hasta cierta longitud, se obtiene que dicha CT es del orden de  $O(\log(n) \cdot \log(\log(n)))$ . Concretamente, esto quiere decir que la función que recibe el inverso de la distancia hasta  $DR = 1$  que se desea obtener (e.d. si la función recibe el valor 4, entonces nuestro DR objetivo es  $1 - 1/4 = 0.75$ ), y devuelve el coste del test suite necesario para alcanzar la DR recibida, se comporta asintóticamente de esa manera.

### **Segundo ejemplo: un implementador típico que ya realizó algunas pruebas sistemáticas**

Esta vez asumimos las hipótesis IH y ICH (ver **Anexo I**). Ahora, nos encontramos con un escenario en el que resta peso el primer paso donde  $f$  produce una salida incorrecta, de modo que la pérdida de peso de cada función es determinada tan sólo por el primer nivel donde ocurre un error. Consecuentemente, el peso de cada máquina (función) viene denotado por la fórmula  $1 - 1/2^{\text{err}(f) - 1}$ , donde  $\text{err}(f)$  indica el nivel del árbol de ejecución en el que se ha encontrado el primer error. Al igual que en el ejemplo anterior, si consideramos que el coste es el número total de *inputs* de la máquina, la CT que se obtiene es del orden de  $O(\log(n) \cdot \log(\log(n)))$ . Observemos el siguiente ejemplo:

Si encontramos un error en el primer nivel del árbol de ejecución, el peso obtenido sería de  $1 - 1/2^{1-1} = 0$ . Si el primer error se encuentra en el segundo nivel, tenemos  $1 - 1/2^{2-1} = 0.5$ . Si se encuentra en el tercer nivel,  $1 - 1/2^{3-1} = 0.75$ . Y así sucesivamente.

### **Tercer ejemplo: un implementador muy competente (así que realizar más pruebas es encontrar una aguja en un pajar)**

Vayamos un paso más allá en la hipótesis del programador competente y asumamos la hipótesis ACH (ver **Anexo I**), de modo que cada máquina contenga como máximo un error (e.d. como máximo una salida de la máquina puede ser errónea). Además, asumamos también DBH, de modo que las funciones que exponen su (único) fallo en un paso posterior

se consideran menos probables. La intuición detrás de esta visión alternativa (y opuesta a la asumida en los dos ejemplos anteriores) es que, si el único error no ha aparecido después de realizar algunas iteraciones largas, entonces es menos probable que aparezca después de realizar iteraciones aún más largas. De hecho, si el error no aparece después de iteraciones muy largas, es más probable que la implementación sea completamente correcta. En esta estrategia, el peso se denota por  $p^{err(f)}$ , donde  $p$  es igual a  $p_r = 1/2^{r+1}$  para cierto  $r > 0$  y,  $err(f)$  indica el nivel del árbol de ejecución en el que se ha encontrado el error. Esta vez, dependiendo de cómo fijemos el valor de  $r$ , la CT resultante varía. Por ejemplo, si tomamos  $r = 1$  y considerando que el coste es el número total de *inputs* de la máquina, entonces la CT obtenida es del orden de  $O(n^{1/r} \cdot \log(n^{1/r}))$ . Fijémonos en el siguiente ejemplo:

Si una función contiene su único error en el primer nivel de ejecución, entonces tendrá un peso de  $(1/2^{1+1})^1 = 0.5$ . Si se encuentra en el segundo nivel, tenemos  $(1/2^{1+1})^2 = 0.0625$ . Y así sucesivamente.

## 2.2. Mutation Testing

### 2.2.1. Finalidad del mutation testing

El objetivo de *mutation testing* (inventado por Richard Lipton [3][4]) es diseñar pruebas de software (e.d. tests) con alta capacidad para detectar errores en la IUT si tales errores existen, es decir, tales que, si la IUT supera dichas pruebas, entonces la IUT tiene una probabilidad más alta de ser realmente correcta. Para poder evaluar a priori la capacidad de un test suite para encontrar errores en cierta IUT, el mutation testing propone simular dichos errores, siendo nosotros mismos los que creamos un conjunto de posibles IUT incorrectas, idealmente representativas de los errores que el testeador cree que la verdadera IUT podría contener. El proceso consiste en modificar ligeramente el programa de varias formas diferentes (cada programa resultante recibe el nombre de *mutante*<sup>4</sup>), y determinar si cada mutante es aceptado o rechazado al ser comparado con su versión original (es decir, observamos si el comportamiento mostrado por ambos para el conjunto de pruebas considerado coincide o no). Si el comportamiento difiere del producido por la versión original, entonces decimos que el mutante ha sido *matado*. Los conjuntos de prueba se evalúan en función del porcentaje de mutantes que son capaces de matar; a mayor porcentaje obtenido, una mayor confianza proporcionarán a nuestro software si nuestro software logra pasar dichas pruebas.

Al mismo tiempo, cabe destacar que los mutantes se basan en operadores de mutación bien definidos que imitan errores de programación típicos (como el uso de un operador lógico incorrecto o el nombre de una variable errónea), y en la práctica permiten descubrir la

---

<sup>4</sup> Un mutante no es más que una copia casi idéntica del programa *original* a la que se ha modificado algún elemento, como por ejemplo una condición lógica, el operador de una operación aritmética, el valor de asignación de una variable, etc. Los mutantes que veremos a lo largo del desarrollo tan sólo tienen cambiado un único elemento, es decir, nunca hablaremos de mutantes con más de una modificación en el código.

necesidad de incluir pruebas valiosas (e.d. una que fuerce a dividir una expresión entre cero). El propósito es ayudar al desarrollador, creando pruebas efectivas capaces de localizar debilidades en los datos o secciones del código a las que rara vez o nunca se tienen acceso durante la ejecución.

### 2.2.2. Visión general

Las pruebas llevadas a cabo en mutation testing se basan fundamentalmente en dos hipótesis. La primera de ellas es la hipótesis del programador competente, la cual establece que la mayoría de los fallos de software introducidas por programadores experimentados se deben a pequeños errores<sup>5</sup> en el código (otra forma de ver esta hipótesis es que consideramos el hecho de que la IUT que esté bien o casi bien más probable que el hecho de que esté muy mal). La segunda hipótesis, llamada *efecto de acoplamiento*, afirma que los fallos simples pueden llegar a formar una cascada o acoplarse para formar otros fallos de mayor criticidad.

Para evaluar la calidad de un determinado test suite (e.d. su capacidad de detección de errores), las pruebas de mutación se realizan seleccionando un conjunto de operadores del programa que pueden ser modificados y, tomando el programa fuente como base, generar otros programas donde cada uno de ellos es el resultado de modificar alguno de dichos operadores en el programa original. El resultado de aplicar un cambio en algún operador es lo que llamamos *mutante*.



Ilustración 4: Generando mutantes

Si el banco de pruebas es capaz de detectar el cambio, entonces se dice que el mutante ha sido matado. Cabe destacar que detectar el cambio no es más que comparar (generalmente el resultado de) la ejecución del programa fuente con la del mutante que se esté probando.

---

<sup>5</sup> Cuando nos referimos a errores de código no estamos hablando de errores que puedan ser detectados en tiempo de compilación. Nos referimos a errores cuyo único impacto es modificar el resultado final que se espera tras la ejecución de un determinado programa.





Ilustración 5: Aplicando el Test Suite

Por ejemplo, consideremos el siguiente fragmento de código en C++:

```

if(a && b){
    c = 1;
}else{
    c = 0;
}

```

El operador de condición `&&` ha sido cambiado por el operador `||`, produciendo el siguiente mutante:

```

if(a || b){
    c = 1;
}else{
    c = 0;
}

```

Ahora, para que la prueba mate a este mutante, se deben cumplir las siguientes tres condiciones:

- La prueba realizada debe *alcanzar* la declaración mutada, es decir, el flujo de ejecución debe pasar por la condición.
- Los datos de entrada de la prueba deberían *infectar* el estado del programa y causar diferentes estados para el programa fuente y el mutante. Por ejemplo, una prueba con `a = 1` y `b = 0` provocaría esto.
- El estado de programa incorrecto (e.d. el valor de la variable `c`) debe propagarse a la salida del programa y ser verificado por la prueba realizada.

El esfuerzo necesario para llevar a cabo pruebas de mutation testing puede ser muy alto incluso para programas pequeños.

### 2.3. Generación de mutantes

Como veremos más adelante, para generar mutantes durante la segunda fase del proyecto tuvimos que buscar algún tipo de software que introdujese cambios en los programas que teníamos que desarrollar para que produjeran salidas erróneas y sirviesen

como simulación al concepto de “definiciones incorrectas de la IUT”. La herramienta que finalmente elegimos fue Pitest [5], no sin antes descartar otras.

En primer lugar, dimos con un *plugin* para Eclipse [6] denominado Muclipse [7]. Esta herramienta permite generar mutantes a través de una interfaz gráfica e incluso elegir qué tipo de modificación se desea realizar sobre un determinado programa (operadores aritméticos y lógicos, tipos de los atributos, condiciones, etc.) para que pase de ser correcto a ser incorrecto. Desafortunadamente, descartamos esta opción debido a tres problemas:

- Ya que este plugin no se encuentra en desarrollo activo desde el 29/09/2011, tan sólo funciona si es instalado en la versión de Eclipse 3.3 o 3.4.
- Sólo se puede ejecutar con JRE 1.6 (la versión actual es JRE 8).
- Sólo funciona mediante la librería *extendedOJ*.

En segundo lugar, tratamos de buscar alguna herramienta que fuese más fácilmente integrable y actual, y dimos con Major [8]. Major nos permitía generar mutantes fácilmente mediante una terminal y ejecutando una serie de comandos. El problema con el que nos encontramos es que tan sólo éramos capaces de generar mutantes de una clase Java. Cuando un proyecto tenía más de una clase, no fuimos capaces de configurar Major correctamente para que no produjese errores de compilación al intentar mutar las clases. Desconocemos si es posible realizar dicha configuración y, debido a que rápidamente encontramos nuestra herramienta definitiva, dejamos de lado Major para centrarnos en Pitest.

### 2.3.1. Pitest

La generación de mutantes se ha hecho de manera automática gracias a Pitest, de ahora en adelante denotado simplemente PIT. Para entender la manera en la que se lleva a cabo la mutación de programas, vamos a profundizar en qué es lo que hace PIT cuando recorre cada línea de código y cómo es capaz de mostrar estadísticas según los resultados obtenidos.

Su funcionamiento se basa en las pruebas de mutación. Lo que PIT hace es ejecutar pruebas contra las copias modificadas del programa original (copias que también se genera él mismo). Cuando el código del programa cambia, debe producir resultados diferentes y provocar que las pruebas fallen. Para modificar el programa original, las mutaciones se siembran automáticamente en el código (siguiendo una serie de reglas que veremos más adelante) y luego se procede a ejecutar las pruebas. Si las pruebas fallan (se logra diferenciar la versión original del mutante), el estado del mutante será *killed*. Si pasa las pruebas (no se logra diferenciar la versión original del mutante), entonces su estado será *lived*. Cabe destacar que PIT realiza sus pruebas y que éstas están basadas en *JUnit* [9].

Cuando PIT finaliza las pruebas genera un informe en un formato fácil de leer que combina la cobertura de código y la información de cobertura<sup>6</sup>.

```

122          // Verify for a "." component at next iter
123  3      if ((newcomponents.get(i)).length() > 0 {
124          {
125          newcomponents.remove(i);
126          newcomponents.remove(i);
127  1      i = i - 2;
128  1      if (i < -1)
129          {
130          i = -1;
131          }
132      }
133  }
```

Ilustración 6: Informe de cobertura de Pitest

Los colores de las líneas de código que aparecen en la ilustración responden a:

- **Verde claro:** líneas de código que ha sido capaz de ejecutar según las pruebas hechas.
- **Verde oscuro:** líneas de código que han sido mutadas y que, al realizar la mutación, han sido *killed*.
- **Rosa claro:** líneas de código que han sido mutadas pero cuyo estado, tras aplicar las pruebas, es *lived*.
- **Rosa oscuro:** líneas de código que no han podido ser ejecutadas. Este último caso suele responder a trozos de código que dependen de bucles u operaciones condicionales para poder ejecutarse.

PIT aplica un conjunto de operadores de mutación, que es configurable por el usuario, al *bytecode* generado al compilar el código del programa original. Por ejemplo, **CONDITIONALS\_BOUNDARY\_MUTATOR**, modificará el *bytecode* generado por la siguiente declaración:

```

if(i >= 0){
    return "foo";
}else{
    return "bar";
}
```

<sup>6</sup> La cobertura de código es una medida utilizada al realizar pruebas de software que determina el grado en el que el código fuente de un programa ha sido comprobado. Sirve para definir la calidad de las pruebas que se llevan a cabo, verificando qué partes del código han sido ejecutadas al menos una vez (cubiertas) por dichas pruebas y qué partes no han sido ejecutadas. Nótese que alcanzar todas las partes del código al menos una vez con las pruebas utilizadas, y haber obtenido resultados correctos para dichas pruebas, no basta para garantizar que el programa correspondiente sea correcto.

Y pasará a ser:

```
if(i > 0){
    return "foo";
}else{
    return "bar";
}
```

Para cada mutación que hace, PIT informará de los siguientes resultados además de los vistos en el ejemplo anterior.

- *Killed y Lived*: hacen referencia a si un mutante ha muerto o sigue vivo.
- *No coverage*: básicamente es lo mismo a estar vivo, excepto que las pruebas que se aplicaron no llegaron al lugar en el que se creó la mutación.
- *Non viable*: una mutación en este estado es aquella que no pudo ser cargada por la JVM (Java Virtual Machine) ya que el *bytecode*, de alguna manera, no era válido. PIT intenta minimizar el número de mutaciones no viables que crea.
- *Memory error*: puede ocurrir debido al resultado de una mutación que aumenta la cantidad de memoria utilizada por el sistema, o puede ser el resultado de la sobrecarga de memoria adicional necesaria para ejecutar repetidamente las pruebas.
- *Run error*: un error en ejecución significa que algo salió mal al intentar llevar a cabo la mutación. Ciertos tipos de mutaciones *non viable* pueden dar como resultado un error en ejecución.

Actualmente, PIT proporciona mutadores que, en su mayoría, son activados por defecto. El conjunto predeterminado puede ser anulado y seleccionar un nuevo conjunto. Estos operadores están diseñados en gran medida para ser estables (e.d. para que nos sean demasiado fáciles de detectar). Aquellos operadores que no cumplen estos requisitos no están habilitados por defecto. Para conocer todos los mutadores disponibles que tiene PIT ver **Anexo II**.

### 3. Desarrollo del proyecto

#### 3.1. Ideas iniciales descartadas

Antes de comenzar a explicar la decisión final que se tomó para realizar el desarrollo, consideramos importante destacar algunas de las ideas iniciales que tuvimos y entender por qué no se llevaron a cabo.

Inicialmente, para hacer las pruebas con un programa que simulase los tres ejemplos nombrados en el artículo *Introducing Complexity to Testing*, pensamos en un proyecto Java<sup>7</sup> que utilizase árboles binarios para representar lo que serían las definiciones de la IUT que vimos anteriormente. Concretamente, consideramos la idea de tener un solo árbol de ejecución que para nosotros representaría tanto la definición correcta de la IUT como los mutantes, que representarían definiciones incorrectas.

Este proyecto Java tenía básicamente dos requerimientos:

- Crear un único árbol de manera que represente conjuntamente a todos los árboles, tanto el árbol original<sup>8</sup> como todos los mutantes que se deseen.
- Realizar dinámicamente el proceso de comparación entre el árbol original y el resto de los mutantes al aplicar las pruebas.

Imaginemos que hay una máquina (el programa o sistema sobre el que aplicar pruebas) que recibe una secuencia de entradas. A continuación, procesa el programa y da el resultado correspondiente de cada entrada.

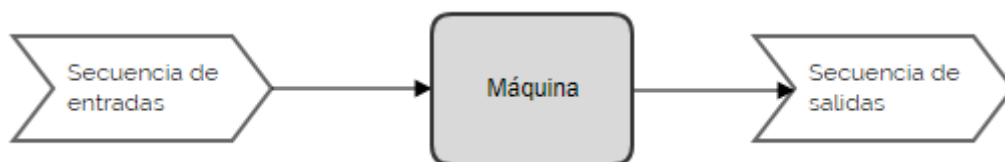


Ilustración 7: Modelo de la primera idea

El programa recibiría como entrada una secuencia de valores 0 y 1, siendo 0 bajar hacia el hijo izquierdo del árbol y 1 bajar hacia el hijo derecho. El resultado (e.d. la secuencia de salidas), estará formado por los símbolos **a** y **b**. Lógicamente, los mutantes causarían resultados en sus salidas que serán distintos a los causados por el árbol original. Así que lo que teóricamente tenemos es un árbol binario original y  $n$  árboles binarios mutantes.

<sup>7</sup> Es importante no confundir “proyecto” (cuando hablemos de forma general de nuestro proyecto de TFG) con “proyecto Java”, que se refiere a un proyecto desarrollado a través de Java, el cual tendrá sus correspondientes clases Java.

<sup>8</sup> Consideraremos «árbol original» a la definición correcta de un determinado programa o, en este caso, estructura de datos. Nótese que un árbol original podría tener el aspecto que fuese, es decir, estadísticamente no importa el valor que almacenen sus nodos, pues siempre va a ser comparado con el resto de mutantes.

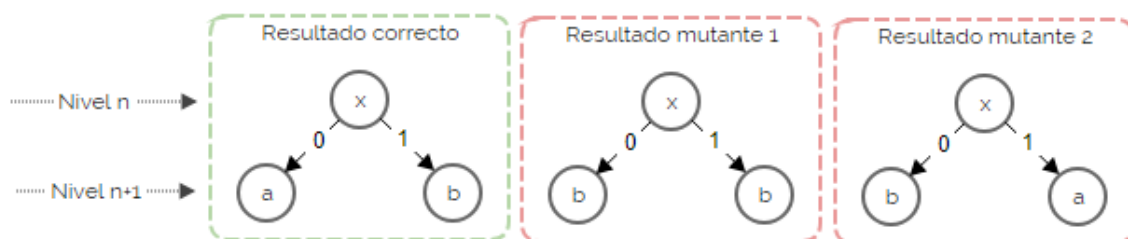


Ilustración 8: Representación teórica de los distintos árboles

Con el fin de ahorrar tiempo de ejecución, decidimos hacer la generación de mutantes de manera dinámica y evitar generar los  $n$  árboles binarios mutantes completos (e.d. con todos sus niveles). La idea consistía en generar un único árbol binario cuyos nodos almacenasen un array con  $n+1$  posiciones ( $n$  mutantes más el árbol binario original).

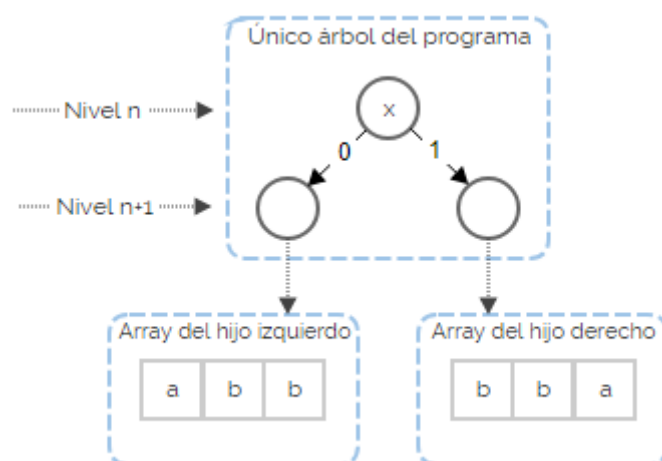


Ilustración 9: Representación real de los distintos árboles

De esta forma, a la hora de aplicar las pruebas tan sólo tendríamos que ir comparando los valores almacenados en las posiciones asignadas a los mutantes con la primera posición del array, donde se almacenan los valores asignados al árbol binario original. Esto nos permite ir haciendo una comprobación por niveles de tal forma que, si estando en el segundo nivel del árbol observamos que el valor de uno de los mutantes difiere con el del árbol binario original, entonces dicho mutante se marca como muerto, y ya no tendría sentido continuar profundizando en sus nodos hijos (e.d. no habría que comprobar todos los arrays por debajo de ese nivel). La siguiente ilustración recoge este último concepto; como vemos, si por ejemplo el *Mutante 1* (segunda posición del array) ha sido matado en el nivel  $n+1$ , entonces sus nodos hijos se ignoran en el nivel  $n+2$  y todos los niveles posteriores.

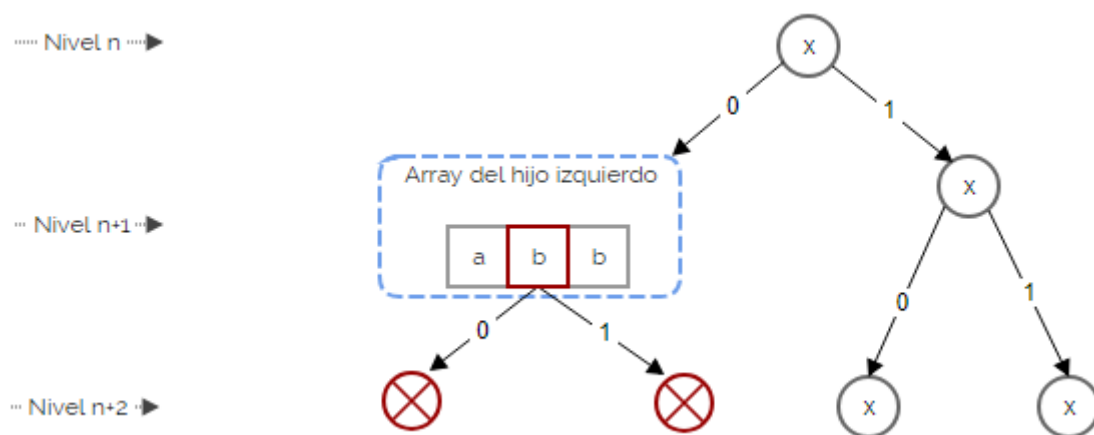


Ilustración 10: Consecuencia al matar un mutante

Desde este punto en el que nos encontramos, surge un grave problema: la memoria RAM disponible se agota con rapidez debido a que el número de nodos del árbol (y, por ende, el de arrays que almacenan dichos nodos) crece de manera exponencial. Observemos que, aunque no se realice la comprobación de los nodos, las estructuras de datos (nodos y arrays que contienen los nodos) ya han sido generadas por la aplicación, luego ocupan espacio en memoria RAM. Además, fijémonos en que esto generaría muchos nodos *zombie*<sup>9</sup> que no servirían para nada más que para ocupar espacio en memoria. Y dependiendo de las pruebas que se apliquen, podría suceder que la memoria ocupada por nodos *zombie* sea incluso mayor que la ocupada por nodos que no son *zombie*.

Para tratar de solventar el problema de espacio en memoria RAM, planteamos dos posibles soluciones:

- (a) Manteniendo esa idea, aprovechar la memoria de disco, haciendo lectura y escrita sobre un fichero para guardar la información necesaria.
- (b) Aprovechar la memoria en disco, pero modificando la idea para que sea más sencilla de llevar a cabo y podamos generar todos los mutantes, de forma que cada mutante sea almacenado en un fichero como un árbol binario, lo que además facilitaría la tarea posterior de aplicar pruebas.

Como veremos durante la explicación en detalle de la Fase I, la opción que finalmente elegimos fue la (b).

### 3.2. Comprendiendo el modelo a desarrollar

Es importante que, antes de comenzar a explicar en profundidad cómo se ha llevado a cabo el desarrollo de las dos fases del TFG y qué es lo que se ha hecho en cada una de ellas, entendamos a grandes rasgos el modelo que se persigue desarrollar y las etapas que

<sup>9</sup> Llamamos nodo *zombie* a aquel nodo (o cualquiera de sus hijos a un nivel mayor de profundidad) cuyo padre tenía un error y ha sido matado por alguna de las pruebas aplicadas.

ha tenido. Es decir, vamos a abordar una visión global del trabajo realizado y a tratar de ver cómo las piezas encajan entre sí.

Si bien es cierto que para la programación del servicio web se han utilizado los lenguajes típicamente empleados en estos casos como Javascript, la librería jQuery y, en cuanto al diseño de la interfaz, HTML y CSS. Para las aplicaciones (tanto en la primera como en la segunda fase) sobre las que se van a aplicar pruebas de testing se ha utilizado el lenguaje de programación Java. Las razones para haber elegido Java son principalmente tres:

- Todas las características que posee Java (ver **Anexo III**).
- Es fácilmente integrable con marcos de prueba como *JUnit* y herramientas de generación de mutantes como Pitest.
- Junto a *C++*, es el lenguaje que más en profundidad hemos estudiado (al menos en *Software e Informática*, nuestros grados).

También nos gustaría hacer hincapié en la idea que forma la base de las pruebas que se van a aplicar, tanto durante la primera fase como durante la segunda. En ningún momento debemos de perder de vista el concepto teórico que vimos cuando hablábamos sobre lo que son las posibles definiciones correctas e incorrectas de una IUT. Como dijimos, lo normal será que una IUT tenga varias definiciones correctas, es decir, que tenga un conjunto de pares correctos y otro conjunto de pares de definiciones incorrectas. Sin embargo, durante el desarrollo del TFG, asumiremos que tan sólo tenemos una posible definición correcta de IUT: el programa original. Veamos la diferencia con la **Ilustración 2** en la que sí aparecían varias definiciones correctas para la IUT.

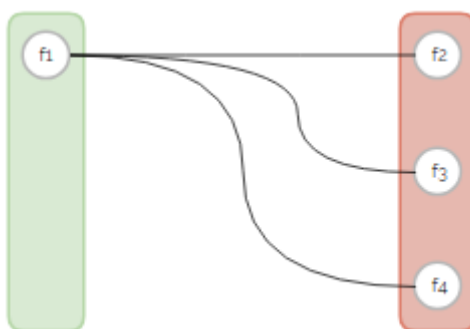


Ilustración 11: Una sola definición correcta de la IUT

El “programa original” es como denotaremos a la definición correcta de IUT, tanto si nos encontramos en la primera fase (en la cual el programa original será un árbol binario) como si nos encontramos en la segunda fase (en la cual el programa original será un programa real, es decir, con condiciones de bifurcación, expresiones lógicas, bucles, etc). ¿Y las definiciones incorrectas de la IUT? Para nosotros, éstas serán los mutantes generados. Nótese que, ante este escenario, podemos obtener el DR de una manera muy simple sin más que dividir el número de mutantes muertos entre el número total de mutantes. Así



pues, cuando un mutante es matado, entonces diremos que hemos sido capaces de distinguir el par {programa original, mutante  $i$ -ésimo}. Por lo tanto, el escenario real con el que nos encontramos es el siguiente:

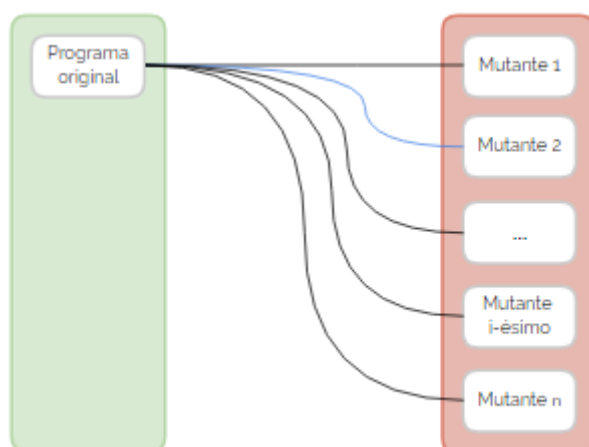


Ilustración 12: Escenario real de definiciones de la IUT

Si observamos la imagen, vemos cómo tan sólo el *Mutante 2* ha sido matado al aplicar pruebas de testing. Luego, si consideramos que hay 10 mutantes, entonces obtendríamos un DR de  $1/10 = 0.1$ .

### 3.2.1. Introducción a la primera fase del proyecto

Como ya hemos avanzado, esta aplicación se desarrollará en forma de proyecto Java dentro del IDE Eclipse para que proporcione una sencilla interfaz por consola que permita al usuario introducir los datos necesarios, llevar a cabo la generación de árboles binarios y poder consultar los resultados a la hora de realizar análisis. Durante la explicación del desarrollo se profundizará más sobre esta idea.

Ya que conocemos el concepto de *mutante*, necesitaremos algún tipo de software que nos permita generar mutantes a los que aplicar baterías de pruebas, ya sean sobre árboles binarios (Fase I) o programas reales (Fase II). En esta primera fase, seremos nosotros mismos quienes dotaremos a la aplicación para que sea capaz de generar el árbol binario original (e.d. lo que sería la definición correcta de la IUT) y las mutaciones deseadas (a través de una serie de opciones que permitan realizar distintos tipos de mutaciones). Durante la explicación de la primera fase veremos en mayor profundidad estos conceptos.

### 3.2.2. Introducción a la segunda fase del proyecto

En esta segunda fase del TFG desarrollaremos un servicio web ejecutado sobre un servidor HTTP Apache [10]. Este servidor se ejecuta gracias a XAMPP [11]. Resumidamente, un servicio web no es más que un tipo de tecnología que utiliza un conjunto de protocolos y estándares para intercambiar datos entre aplicaciones. La razón principal por la que queremos desarrollar un servicio web es que nos permite utilizar una

interfaz (que puede ser usada a través del protocolo HTTP) para interactuar con el programa Java que se ejecutará “por debajo” del servicio web. Asimismo, podemos tratar los datos de salida y mostrarlos de forma que sea más agradable para el usuario (mostrar tablas ordenadas por campos, dibujar gráficas de CT en base a diferentes parámetros, importar proyectos, etc.). Por supuesto, podríamos haber obviado esta parte y simplemente manejar el proyecto desde Eclipse, pero consideramos que, si bien la presentación es mucho menos importante que los resultados en sí, también es una pieza clave a la hora de mostrar los análisis y las conclusiones obtenidas. Al mismo tiempo, los datos recibidos y mostrados por el servicio web serán almacenados en una base de datos MariaDB [12], cuyo esquema veremos en detalle más adelante.

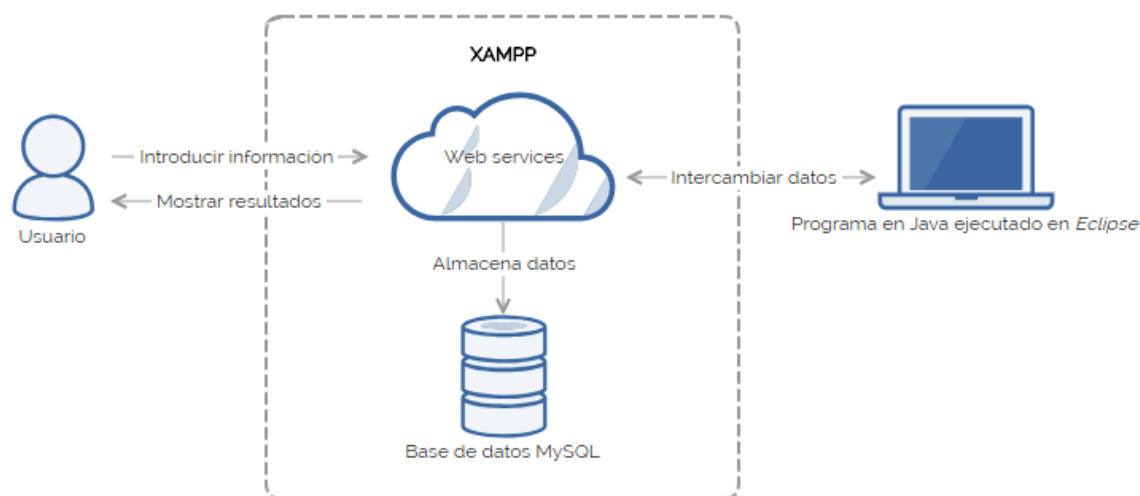


Ilustración 13: Modelo introductorio a la fase II

En cuanto a la generación de mutantes durante esta segunda fase, la idea de mutar programas se vuelve bastante más complicada. Al fin y al cabo, lograr modificar una estructura de datos como un árbol binario es relativamente sencillo, pero crear copias de programas en los que, de alguna forma, el usuario pueda elegir qué elemento cambiar para crear un mutante, ya no resulta tan sencillo. También queremos mencionar los dos tipos de pruebas que se llevarán a cabo con programas Java.

- En primer lugar, las pruebas se aplicarán a un programa creado manualmente, el cual tiene una serie de clases que serán testeadas.
- En segundo lugar, explicaremos el desarrollo de un programa Java que, a su vez (y gracias a una serie de parámetros), es capaz de generar otros programas Java de manera automática.

La información de los proyectos Java sobre los que se harán pruebas será almacenada en la base de datos, de forma que podremos consultar qué tests se han aplicado a qué proyectos Java y obtener estadísticas sobre los resultados tras las pruebas de testing (número de mutantes muertos, DR obtenido, tiempo de ejecución consumido por las pruebas, etc.).

### 3.3. Fase I: mutaciones en árboles binarios

Pasamos a abordar lo que ha sido el desarrollo de la primera fase del TFG. Recordemos que el objetivo de esta primera fase es simular los tres ejemplos del artículo *Introducing Complexity to Testing* a través de una aplicación desarrollada en Java.

Con respecto a la representación de los árboles binarios, no se ha implementado ninguna estructura en Java tal y como pudiera haber sido la clase *TreeMap* definida en *java.util*. Por una mayor comodidad a la hora de manejar mutantes y, sobre todo, al momento de llevar a cabo la generación de los mismos, se optó por utilizar la clase *ArrayList* (también disponible desde *java.util*), la cual proporciona una serie de métodos (ver **Anexo IV**) que facilitan enormemente la tarea de generación de mutantes. Dicho lo anterior, se estableció que la forma de representación sería la de un árbol binario almacenado en un array tras haber hecho un recorrido por niveles sobre todos los nodos del árbol. Veamos cómo se lleva a cabo este proceso.

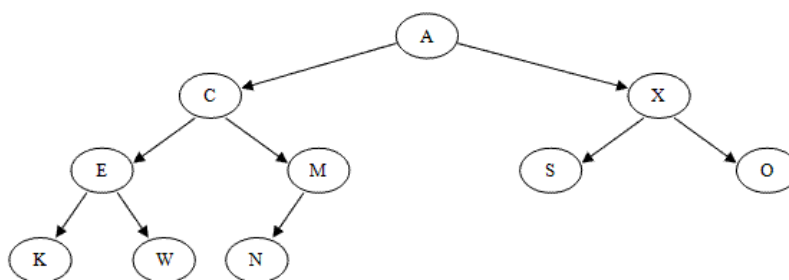


Ilustración 14: Representación de un árbol binario

Observemos cómo el nodo raíz, cuyo valor almacenado es **A**, se almacena en la posición **[0]** del array. Continuando, pasamos al segundo nivel y, de izquierda a derecha, nos encontramos con los valores **C** y **X**, los cuales insertamos en las posiciones **[1]** y **[2]** del array. Análogamente, completamos el resto de las posiciones hasta llegar a la última hoja del árbol.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
A	C	X	E	M	S	O	K	W	N

Ilustración 15: Representación en array de un árbol binario por niveles

Cabe destacar que esta forma de almacenar los valores de los nodos tan sólo es válida si se trata de árboles binarios completos (árboles que poseen todos sus nodos hasta un determinado nivel) o incompletos en su último nivel.

Como veíamos anteriormente, nuestra aplicación también contiene una sola definición de IUT correcta. Esta definición la representaremos como un árbol binario que contiene un

valor 0 almacenado en todos sus nodos. Al mismo tiempo, una definición incorrecta de la IUT (mutante) será representada como un árbol binario que tiene, al menos, un valor 1 almacenado en uno de sus nodos. Así podemos distinguir definiciones correctas e incorrectas sin más que ir comparando nodo a nodo (posición a posición del array) cada uno de los mutantes con el árbol original hasta encontrar el (o los) nodo (nodos) con valor 1 almacenado. De esta forma, perseguimos que los tres ejemplos del artículo *Introducing Complexity to Testing* sean simulados empíricamente con la mayor similitud posible a los conceptos teóricos vistos.

### 3.3.1. Implementación y funcionamiento

Para llevar a cabo esta primera fase del proyecto hemos desarrollado una aplicación en Java que permite generar cierto número de mutantes y de tanta profundidad como se desee, aplicar distintos tests, obtener el coste de los mismos y consultar el DR resultante. El proyecto Java se encuentra alojado en *Github* [13] y puede consultarse en el siguiente [enlace](#).

#### Construcción de árboles

Tengamos en cuenta que el número total de mutantes a generar crece de manera doblemente exponencial con respecto a los niveles de un determinado árbol. El cálculo se realiza de la siguiente forma: primero debemos calcular el número de nodos de un árbol, el cual, viene dado por la fórmula  $2^{\text{niveles}} - 1$ , donde *niveles* es el número de niveles del árbol. Una vez obtenido el número de nodos, podemos obtener el número total de mutantes que pueden ser generados recurriendo a la fórmula  $2^{\text{nodos}}$ .

Antes de continuar debemos hacer una aclaración importante. En cuanto a las pruebas que realizaremos, diremos que aplicar un *test* hasta cierta profundidad será realizar la comprobación de una sola rama del árbol de ejecución de la IUT (e.d. recorrerla desde la raíz hasta algún nodo) mediante una secuencia de *inputs*. Por otro lado, un *test suite* es aquella batería de pruebas que contiene todos los *tests* hasta una cierta profundidad del árbol (e.d. todas las secuencias de *inputs* posible de cierta longitud). Además, consideramos que el árbol que sólo tiene su raíz tiene nivel 1 (la raíz representa el nivel 1 del árbol).

Por ejemplo: un *test suite* podría llegar hasta el segundo nivel de un determinado árbol, lo que sería equivalente a aplicar 2 *tests* (pues sólo hay dos caminos). Pero un test suite de tres niveles sería equivalente a aplicar 8 tests, y así sucesivamente.

Por último, cabe destacar que, aunque un test suite es un conjunto de secuencias de *inputs*, alternativamente se puede ver como el árbol (etiquetado sólo con *inputs*, no son *inputs* y *outputs*) que induce.

Avanzando en las pruebas, estamos especialmente interesados en los tres casos nombrados durante la explicación del artículo *Introducing Complexity to Testing* y en comprobar empíricamente si la complejidad de testing es la misma que la que aparece en los ejemplos

teóricos, lo cual es lo esperable. Una vez ejecutada la aplicación y tomando el árbol original, se le presentan al usuario, a través de la consola, cuatro posibilidades a la hora de llevar a cabo la generación de mutantes.

1. *Generar todos los mutantes hasta un determinado nivel:* tal y como si de una tabla de verdad se tratase, esta opción genera todos los mutantes posibles del árbol original hasta un cierto nivel. La función encargada de llevar la generación es **generateTotalMutants()**. Cabe decir que esta opción no resulta especialmente interesante a la hora de realizar pruebas por dos sencillas razones.
  - a. La primera de ellas es el gran número de mutantes que se generan. Estamos hablando de que, con árboles de tan sólo cuatro niveles, se generan  $2^{15}$  mutantes, es decir, 32.768. Éstos son generados en aproximadamente unos tres minutos con un ordenador cuyo procesador es un *i7* de gama media. El problema es que, si intentamos profundizar tan sólo un nivel más, obtenemos un total de  $2^{31}$ , es decir, 2.147.483.648 mutantes, lo cual se vuelve intratable con los medios disponibles.
  - b. La segunda razón viene dada por la forma en la que se lleva a cabo la generación; como queremos generar todos los mutantes, esta generación se realiza de un modo mecánico, así que ni siquiera haría falta simular los mutantes (con hacer los cálculos necesarios se podría obtener el DR para cualquier test que se quiera aplicar). Cuando nos referimos a un “modo mecánico” queremos decir que la generación de mutantes sigue un patrón (de hecho, se generan de la misma forma que una tabla de verdad), luego al realizar pruebas, podemos saber (según dicho patrón) la cantidad de mutantes generados y muertos (lo que ya es suficiente para calcular el DR).
2. *Generar algunos mutantes hasta un determinado nivel con errores donde todos restan peso:* se le pide al usuario que introduzca el número de niveles que desea que tengan los mutantes y el número de mutantes a generar. Para cada nodo de cada mutante, se introduce un 0 (no hay error) o 1 (hay error). Su peso final será 1 menos la suma de todos sus errores.
3. *Generar algunos mutantes hasta un determinado nivel donde sólo el primer error resta peso en cada mutante:* análogamente al ejemplo anterior, se pide número de niveles y número de mutantes a generar. La diferencia, como vemos en la **Ilustración 21**, es que tan sólo se resta peso el primer error encontrado (profundizando por niveles y siempre de izquierda a derecha).
4. *Generar algunos mutantes hasta un determinado nivel con un solo error:* como en los ejemplos anteriores, volvemos a pedir niveles y mutantes a generar. Si nos fijamos, esta vez los mutantes tan sólo tienen un error, el cual se introducirá siendo

más probable en niveles inferiores. Con lo cual, para calcular el peso, basta con encontrar ese error, ver el peso que tiene en función del nivel en el que se encuentre, y restar a 1 dicho peso.

### Construcción de los tests

Cuando un test se aplica, éste lo hace comenzando desde los arcos que van a su hijo izquierdo e hijo derecho. Mientras un test está aplicándose, se comprueba, en cada arco, si hay un error, es decir: si la salida es 1, entonces se trata de un mutante, con lo cual puede ser matado. Si esto ocurre, se marca como *killed*, añadiendo su peso correspondiente para, posteriormente, obtener el DR. Nos encontramos con dos tipos de test suites en función de cómo son aplicados.

1. *Test suite completo hasta cierto nivel*: un test suite que comprueba todo el comportamiento de la IUT para todas las secuencias de entrada de cierta longitud, es decir, todas las ramas del árbol de ejecución hasta cierta profundidad. Recorre arcos hasta que llega al nivel elegido por el usuario o hasta que se encuentra con el último nivel del árbol. Este test suite, al igual que los mutantes, es almacenado en un array para una mayor comodidad a la hora de manejarlos. El coste para este tipo de test suite viene dado por el número de arcos del árbol de ejecución recorridos por el test suite, lo cual es igual a la profundidad alcanzada por los tests del test suite multiplicado por el número de secuencias de inputs con longitud igual a dicha profundidad que pueden formarse. De forma general, un test suite de  $n$  niveles contiene  $n \cdot \text{hojas}$  tests, donde  $n$  indica el nivel de profundidad que alcanza el test suite y *hojas* indica el número de nodos hoja que hay en dicho nivel. Puesto que esta comprobación se realiza por iteraciones, tomemos como ejemplo el siguiente mutante y veamos cómo, en cada iteración, se ahonda un nivel más hasta, o bien finalizar donde el usuario pidió, o bien llegar hasta el último nivel del mutante. Nótese que la **Ilustración 14** se utilizó con el objetivo de explicar cómo se almacenan los árboles. Pero estos árboles, recordemos, son completos, tal y como se muestra en la siguiente ilustración.

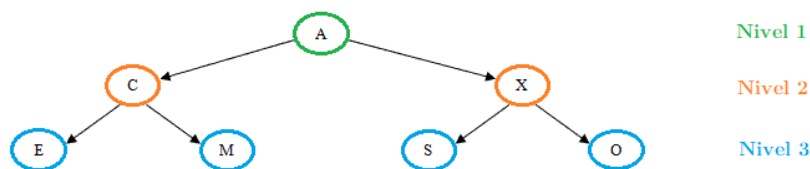


Ilustración 16: Aplicando test suite

2. *Test suite formado por un único test*: el objetivo de un *test* es establecer un camino a testear, tomando la decisión de hacia donde ir (izquierda o derecha) cada vez que llegamos hasta un nodo. Una vez que el usuario introduce la longitud del *test* (hasta qué nivel desea llegar al explorar el camino), se le pedirá que vaya describiendo el camino, pulsando **a** si quiere ir hacia la izquierda o **b** si quiere ir hacia la derecha.

Al igual que en el caso anterior, el *test* es almacenado en un array. El coste viene dado por el número de arcos que atravesamos durante el camino. Por lo tanto, el coste del *test* es igual a la longitud del *test* (e.d. el nivel hasta el que hemos profundizado). Tomando el ejemplo anterior, veamos el camino  $\{aa\}$ .

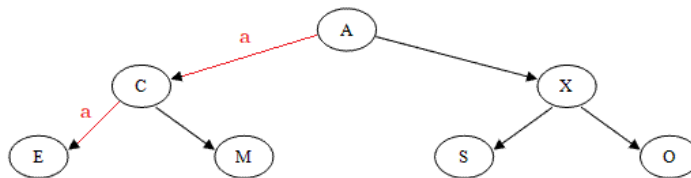


Ilustración 17: Aplicando un test

### 3.3.2. Los tres escenarios a testear

Llegados a este punto, estamos interesados en tres casos sobre los que realizar pruebas de mutación para ver qué CT supone testearlos hasta lograr una cierta confianza en su corrección. Para las pruebas se han generado, en cada caso particular, 100.000 mutantes con 6 niveles de profundidad cada uno, lo que equivale a mutantes con 64 nodos. Consideramos importante destacar que la obtención del DR en los tres escenarios es de la misma forma: peso total de los mutantes muertos dividido entre peso total de todos los mutantes (el concepto de peso se detalla a continuación).

Para cada uno de los tres ejemplos se han adjuntado las gráficas de DR obtenidas en las que se relaciona el coste del test suite aplicado con el DR alcanzado al realizar las pruebas. Antes de entrar en detalle en la simulación de cada ejemplo, es importante mencionar que las pruebas realizadas han sido llevadas a cabo aplicando test suites completos a diferentes niveles de profundidad. La razón es que la CT que aparece en el artículo viene calculada para el caso en que se consideran todas las secuencias de inputs posible hasta cierta longitud (e.d. el conjunto de todos los *tests* posibles hasta cierta profundidad). Por ahora, tan sólo veremos las gráficas para comprobar cómo el DR converge a 1 y qué esfuerzo nos supone en cada uno de los tres ejemplos. Más adelante, en el apartado de **Análisis de la Fase I**, nos centraremos en hacer un estudio sobre la CT para cada caso.

La forma en la que se asignan los pesos y el procedimiento de cálculo es el mismo que el mencionado en los tres ejemplos del artículo (véase el apartado **Introducing Complexity to Testing**).

#### Primer ejemplo: árboles donde todos sus errores restan

En este caso simularemos el primero de los ejemplos del artículo *Introducing Complexity to Testing*. Nos encontramos en un escenario en el que cada uno de los mutantes (recordemos, posibles definiciones incorrectas de la IUT en el ejemplo del artículo), pueden contener en sus nodos un valor 0 o 1. Referente a los pesos de cada mutante la resta que se realiza nunca será negativa, pues en un mutante cuyos nodos almacenan todos ellos un valor de 1, el valor total a restar será como máximo 1, y como resultado de la resta,

obtendremos un 0. Ya que nuestro objetivo es conseguir sumar un mayor número de pesos de mutantes muertos (para obtener un mayor DR al finalizar las pruebas), fijémonos en cómo, en el caso anterior de un mutantes con peso 0, no obtendríamos ningún “beneficio” al matar al mutante (lógico, pues no obtenemos “recompensa” alguna por lograr matar un mutante totalmente incorrecto).

Al igual que existe la improbable situación de que nos encontramos con un mutante completamente incorrecto, también existe la improbable situación de que nos encontremos con un mutante completamente correcto (sin ningún error). Esta situación hemos optado por ignorarla debido a dos razones:

- La consideramos tan improbable utilizando árboles de 6 niveles (e.d. árboles de 64 nodos donde una repetición sucedería cada  $2^{64}$  árboles) que, aunque sucediese, esto no haría cambiar notablemente el DR obtenido (pues sólo se darían unos pocos casos aislados). Así que simplemente asumimos que, si esto ocurre en algún árbol, entonces sumaremos 1 más al peso total de mutantes muertos (lo cual no es mucho considerando que hay 100.000 mutantes).
- En caso de querer solucionar este pequeño problema, nos llevaría mucho tiempo de ejecución ir comprobando mutante a mutante y averiguar si se trata de un mutante completamente correcto. Recordemos que, en el segundo y tercer ejemplos simulados, cuando se está comprobando un mutante y se encuentra un error, el testeo de dicho mutante finaliza y se procede a comprobar el siguiente mutante. Pero si quisiéramos comprobar todos los arcos, entonces el tiempo de comprobación aumentaría considerablemente.

Para obtener los datos del experimento, se han aplicado distintos test suites que llegan hasta los niveles de profundidad 2, 3 y 4 de los mutantes. A nivel 5 ya se alcanza un DR = 1 debido al *efecto frontera* (véase el **Anexo V**), por lo que, con los medios disponibles, tan sólo hemos logrado obtener tres puntos en la gráfica.

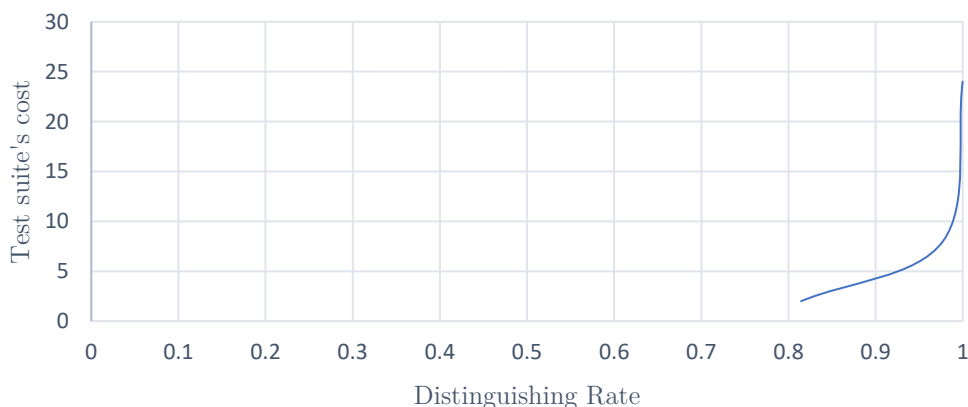


Ilustración 18: DR primer ejemplo aplicando test suite hasta nivel 4



¿Cómo es posible que se alcance un DR tan cercano a 1 con costes tan pequeños? Tengamos en cuenta que un test con sólo un *input* divide a los mutantes “por la mitad”: al asumir un *output* binario por cada *input* binario, la mitad de los mutantes responderá ante el primer *input* lo mismo que la máquina correcta, así que un test con un input mata a la mitad de mutantes, y llegar a  $DR = 0.5$  es muy poco costoso.

### Segundo ejemplo: árboles donde sólo el primer error encontrado resta

Pasamos a un escenario muy similar al del ejemplo anterior. Se ha procedido de la misma forma a calcular el DR y se ha asumido el improbable caso de encontrar un mutante completamente correcto. La forma en la que se realiza el cálculo para este ejemplo hace que los pesos de estos mutantes tiendan a ser relativamente más altos respecto al caso anterior (pues se le resta a 1 una cantidad menor).

Al igual que en el caso anterior, observamos que llegar a un DR cercano a 1 conlleva un coste muy bajo. De nuevo, se han aplicado test suites hasta niveles de profundidad 2, 3 y 4 consecutivamente a los 100.000 mutantes generados, obteniendo la siguiente gráfica de DR:

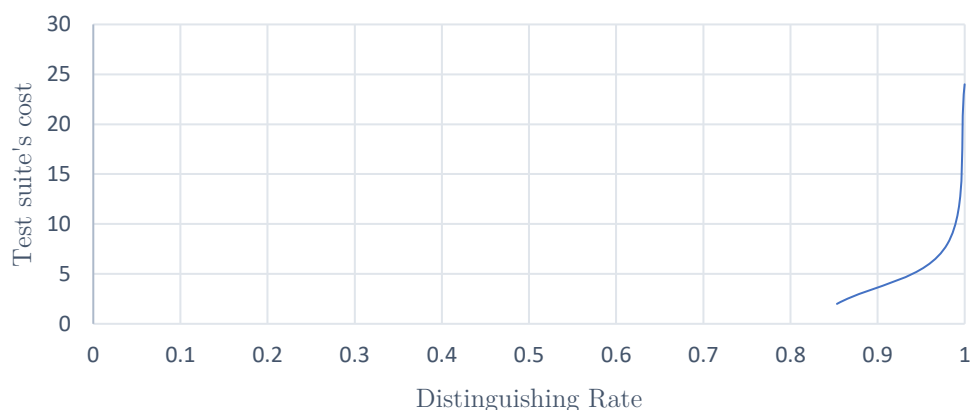


Ilustración 19: DR segundo ejemplo aplicando test suite hasta nivel 4

### Tercer ejemplo: árboles donde su único error resta

Pasamos al último de los escenarios cuyos mutantes, de forma aleatoria, tienen un solo error. Como vimos durante la explicación del tercer ejemplo del artículo, según qué valor tomemos para  $r$ , la CT resultante variará (los análisis de CT se verán más adelante). En nuestra simulación empírica, hemos tomado el valor  $r = 1$ . En cuanto al improbable caso de tener un mutante completamente correcto, aquí nunca podrá darse ese caso porque se fuerza a meter un error según las probabilidades teóricas que vimos. Estos mutantes, al tener sólo un error, tienden a tener pesos relativos altos, más que incluso en el segundo ejemplo debido a que es más probable que el error se produzca en los niveles más profundos del árbol. No obstante, en este caso la principal diferencia, en términos de la dificultad de encontrar errores a través del testing, reside en que los errores de los mutantes son mucho

más difíciles de detectar, ya que sus árboles de ejecución contienen un único arco incorrecto. Es por eso que aplicando el mismo esfuerzo de testing (e.d. usando test suites de igual coste), la DR alcanzada tiende a ser considerablemente menor comparado con los dos ejemplos anteriores, lo que nos hace llegar a un  $DR = 1$  en niveles más profundos que en los dos primeros ejemplos.

Concretamente, esta vez se han generado 100.000 mutantes de 7 niveles cada uno, pudiendo obtener un  $DR < 1$  para los 6 primeros niveles tras aplicar un test suite, lo que nos ha permitido obtener más puntos en la gráfica respecto a los anteriores escenarios simulados (recordamos que para nosotros la raíz del árbol es el nivel 1).

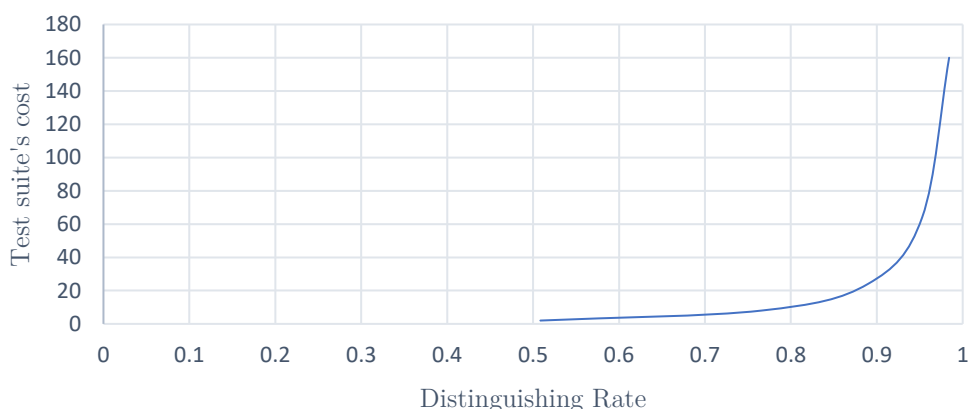


Ilustración 20: DR tercer ejemplo aplicando test suite hasta nivel 6

### Cómo se almacenan los mutantes en ficheros

Consideramos importante destacar cómo los mutantes son almacenados en ficheros de texto para su posterior lectura al aplicar un test o test suite. Todos los mutantes, cuando se almacenan, tienen la siguiente estructura: comienzan con un número decimal que establece el peso del mutante (ya realizada la resta según cada caso particular de los tres vistos anteriormente). A continuación, se presenta el array de  $2^n$  posiciones, donde  $n$  es el número de niveles que tiene el mutante. En la primera posición del array (posición `[0]`), encontramos “basura”. En esta posición simplemente hemos introducido un 0 ya que no nos es útil. La razón es porque un árbol en realidad tiene  $2^n - 1$  nodos, así que, por temas de control de índices, hemos optado porque esta posición inservible sea la primera del array en vez de la última. Siguiendo encontramos el árbol en sí, es decir, esos  $2^n - 1$  nodos nombrados anteriormente. Por último, un `-1` que indica la finalización de lectura. Los siguiente dos ejemplos ilustran esta idea.

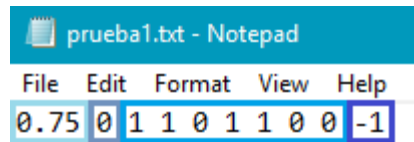


Ilustración 21: Mutante del segundo ejemplo

Tenemos un mutante con muchos errores introducidos, pero en el que sólo se tiene en cuenta el primero. Como dijimos anteriormente, la raíz no tiene penalización, luego nos queda que el primer error encontrado pertenece al segundo nivel, por lo tanto, su peso sería:  $1 - 1/4 = 0.75$ .



Llegados a este punto, vamos a entrar en profundidad con cada uno de estos procesos, ver el orden en el que se ejecutan, para qué sirven cada uno de ellos, por qué todos son necesarios y qué es lo que cada uno de ellos aporta durante cada una de sus etapas de ejecución. Ya que durante la explicación de cada uno de ellos podríamos perder la noción de en qué punto nos encontramos, adjuntamos una ilustración donde se muestra la ejecución natural que se lleva a cabo desde que el usuario comienza a interactuar con la interfaz web hasta que visualiza los resultados de las pruebas aplicadas. Si fuese necesario, el lector puede volver a este punto para tener una visión general de los procesos e ir a cualquiera de ellos nuevamente.

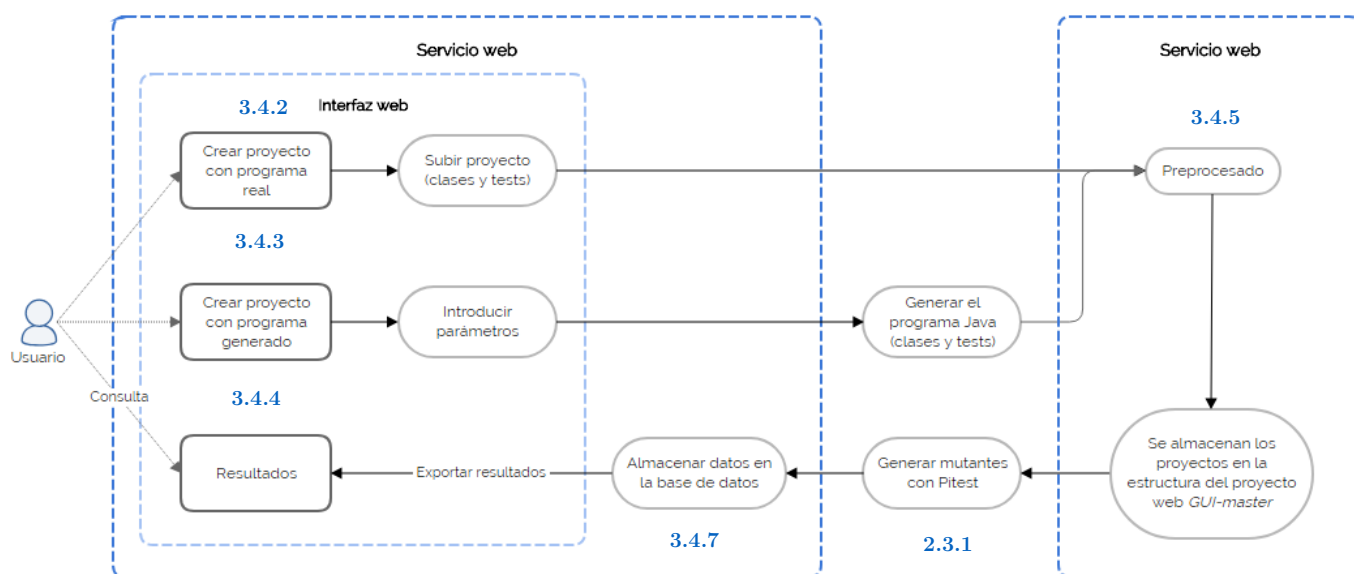


Ilustración 23: Ejecución natural del proyecto

### 3.4.2. Interfaz de usuario

Cuando accedemos a la interfaz web de la aplicación nos encontramos principalmente con tres opciones. Vamos a ver por encima en qué consiste cada una de ellas. Estas opciones se explicarán detalladamente durante los próximos puntos del documento, pero es importante que, antes de entrar en cada uno de ellos, tengamos una visión global de lo que ofrece nuestro servicio web.

1. *Resultados*: aquí encontraremos una tabla donde aparecen todos los proyectos Java que se han cargado en el servicio web. De cada uno de estos proyectos podemos consultar gráficas sobre la CT que ha supuesto testearlos, qué tests se han aplicado y número de mutantes. Por último, dos funcionalidades muy útiles son las de los botones “Mostrar gráficas de varios proyectos” y “Comparar dos proyectos”. En la primera de ellas podemos hacer una búsqueda por nombre y comparar todos los proyectos cuyos nombres coincidan con la búsqueda. Y en la segunda opción podemos comparar dos proyectos por sus IDs.

Home
Resultados
Crear proyecto con programa real
Crear proyecto con programa generado

## Resultados

Nombre del proyecto \*

Comparar gráficas de varios proyectos
Comparar dos proyectos

Id	Name	NumTestsFiles	TotalTime	NumMutants	Avg Killed	Avg Percent	
3634	Proyecto_entrega	2	15.37	311	20.5	6.5	Estadísticas
3633	01234_2_Atr_AnidFor__12_ExprsSeg__12_	1	208	2618	374	14	Estadísticas

Ilustración 24: Vista de la opción Resultados con dos proyectos

2. *Crear proyecto con Programa real*: permite subir un proyecto Java para ser testeado. En esta opción, el usuario debe seleccionar los ficheros de las clases Java que desea testear y también los ficheros de los tests que van a testear dichas clases. Además, se le permite elegir los tipos de mutaciones que se aplicarán a su proyecto Java (en rojo, las que no se aplican por defecto).

Seleccione los tipos de mutaciones: \*

INCREMENTS	<input type="checkbox"/>
MATH	<input checked="" type="checkbox"/>
CONDITIONALS_BOUNDARY	<input checked="" type="checkbox"/>
NEGATE_CONDITIONALS	<input checked="" type="checkbox"/>
INVERT_NEGS	<input checked="" type="checkbox"/>
RETURN_VALS	<input type="checkbox"/>
VOID_METHOD_CALLS	<input type="checkbox"/>
CONSTRUCTOR_CALLS	<input type="checkbox"/>
INLINE_CONSTS	<input type="checkbox"/>
NON_VOID_METHOD_CALLS	<input type="checkbox"/>
REMOVE_CONDITIONALS	<input type="checkbox"/>
EXPERIMENTAL_MEMBER_VARIABLE	<input type="checkbox"/>
EXPERIMENTAL_SWITCH	<input type="checkbox"/>

Ilustración 25: Tipos de mutaciones a aplicar sobre proyectos Java

3. *Crear proyecto con Programa generado*: se le propone al usuario generar su propio proyecto Java gracias a una serie de parámetros que van a definir la estructura que tendrá (número de condiciones, bucles, anidación, etc.). Este proyecto, al igual que el anterior, va a contener clases Java y los tests que se aplicarán sobre esas clases, sólo que esta vez, tanto las clases como los tests, se generan de manera automática (todo ello gracias a una aplicación Java que se ejecuta *por debajo*, la cual recibe el valor de los parámetros y lleva a cabo la generación automática de clases). Análogamente, también se podrá elegir los tipos de mutaciones a aplicar sobre el proyecto Java generado.

### 3.4.3. Crear proyecto con Programa real

Como hemos avanzado anteriormente, el objetivo que se persigue en este escenario es desarrollar dos programas escritos a mano que sean lo más realistas posibles y sobre los que logremos obtener diferente CT variando algunas partes de la estructura de los programas.

Partimos de la idea de que, si un programa tiene un alto grado de anidación, la función que representa la CT podría crecer muy rápidamente. En otras palabras, con un alto nivel de anidación habrá más instrucciones que requieran un alto número de condiciones para ser ejecutadas, que por tanto podrían ser alcanzadas únicamente por unos pocos tests (en lugar de la mayoría de los tests). Poder encontrar potenciales errores en dichas instrucciones requerirá, en principio, aplicar tests suites más amplios, lo que aumentará el coste del testing. Por el contrario, si un programa tiene un bajo grado de anidación, la función que representa la complejidad de testing podría crecer más lentamente. De hecho, esto es lo que se espera que ocurra, pero además de intentar corroborarlo, estamos interesados en saber cómo es la diferencia (e.d. cómo varía la CT a medida que vamos aplicando pruebas) al testear programas con dichas diferencias estructurales.

Recordemos que para la generación de mutantes en cualquier proyecto Java se ha utilizado la herramienta Pitest. A la hora de realizar las mutaciones de estos dos programas, se ha usado la misma política de mutación, es decir, los mutantes se han hecho en las mismas partes de código para los dos programas (se han dejado por defecto las mutaciones que Pitest hace y que vimos durante la explicación de la herramienta). Además, las pruebas aplicadas se hacen de la misma forma para los dos programas: primero se aplica el primer test, que cubre cierta parte del código. Luego se aplica el primer test y un segundo test, por lo que se cubre una parte mayor de código (la cobertura de código aumenta). Y así sucesivamente hasta aplicar  $n$  tests, siempre tratando de buscar una mayor cobertura de código. Concretamente, para las pruebas realizadas hemos decidido aplicar un máximo de siete tests.

Por supuesto, todos los tests han sido programados también manualmente, por lo que todos ellos son distintos, tanto en el primer programa como en el segundo. Precisamente por ese motivo, los tests no logran alcanzar un DR relativamente alto; hay que tener en cuenta que los proyectos que se van a testear tienen cierta complejidad, y el esfuerzo que supondría diseñar un test suite que alcanzase un DR relativamente alto sería muy costoso. Además, los tests han sido diseñados intuitivamente para cubrir la mayor cantidad de cobertura de código posible basándonos en nuestros conocimientos actuales (para más información, consultar el **Anexo VII**).

Siguiendo con esta idea, pasamos a explicar los dos programas reales (e.d. escritos a mano, no diseñados automáticamente) que se han desarrollado y a ver sus características.

### **Programa para crear personajes**

Este primer programa, al que llamaremos *Programa personajes*, consiste en crear un personaje en abstracto y añadirle atributos, como por ejemplo su nombre, edad, qué coche tiene, donde vive y cómo es su casa, etc. Lo que ocurre es que, antes de crear un personaje y añadirle sus atributos, también hay que crear dichos atributos. Por ejemplo: el coche tiene ruedas, asientos, carrocería, etc.

Como muchos de los objetos que hay son figuras, éstos serán representados como figuras 3D. Continuando con el ejemplo, un coche podría ser un rectángulo con cuatro círculos. Es más, podemos ir un paso más allá y establecer que los objetos 3D están formados por objetos 2D. De nuevo, un cubo puede estar formado por un cuadrado y una determinada profundidad.

Llegados a este punto, tratamos todas las *piezas* mencionadas como clases Java y la construcción de cada objeto está definida por la constructora de su clase correspondiente. Las características de este programa son las siguientes:

- Las estructuras de las clases son simples, pero debido a cómo se construye el personaje, hay entre dos y tres niveles de herencia para construir los atributos.
- No hay muchas operaciones aritméticas ya que la mayoría son asignaciones.
- Posee expresiones de bifurcación, pero con un bajo grado de anidación con el fin de que la complejidad de testing crezca lentamente.

Después de aplicar un test tras otro hasta llegar a los siete, hemos obtenido una cobertura de código cercana al 100%. Dentro de esta cobertura es donde Pitest ha realizado todas sus mutaciones. Como veremos a continuación, con los tests aplicados (cuyos costes individuales son de 1) hemos logrado alcanzar un DR muy próximo a 0.5, lo cual no es nada despreciable teniendo en cuenta las dimensiones de este programa para crear personajes. Asimismo, estos tests han sido desarrollados manualmente, por lo que a medida



que queremos acercarnos a un  $DR = 1$ , el esfuerzo y dedicación en diseñar los tests se incrementa considerablemente. Pensamos que dedicar demasiado esfuerzo para lograr un DR cercano a 1 está fuera del alcance nuestros objetivos, por eso, hemos tratado de mantener un equilibrio entre el esfuerzo dedicado y el DR obtenido. Si el lector desea profundizar más sobre el código del programa, puede ser consultado en el siguiente [enlace](#).

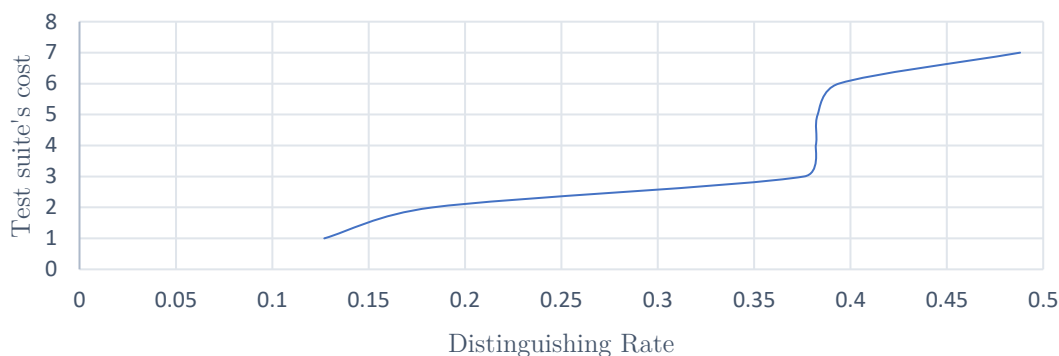


Ilustración 26: DR *Programa personajes* aplicando diez tests

Queremos destacar que debido al diseño manual de los tests es posible que surjan casos en los que un test no aporta un mayor DR (e.d. no logra matar más mutantes de lo que el anterior tests aplicado mató) pero, obviamente, sí incrementa el coste de las pruebas. Esto es precisamente lo que ocurre en la ilustración anterior. Como vemos, entre los tests 3 – 6, la cantidad de DR es prácticamente inapreciable. De nuevo, diseñar tests en los que aseguremos que el DR siempre crece tras ser aplicados consecutivamente, podría llegar a suponer un alto esfuerzo en proyectos como este en los que tenemos una multitud de clases Java, esfuerzo que no entra dentro de nuestros objetivos.

### Programa para consultar expresiones matemáticas

Este segundo proyecto, al que llamaremos *Programa math*, tiene una estructura más “sencilla” que el anterior, y tan sólo consta de tres clases:

- *Vista*: se encarga de interactuar con el usuario y enviar la información que le proporcione a la clase *ControlMath*.
- *ControlMath*: intermediario entre la clase *Vista* y *Ecuaciones*. Se encarga de pasar la información que recibe entre ambas.
- *Ecuaciones*: tiene los datos referentes a las ecuaciones matemáticas que utiliza el programa.

El programa está formado por varias funciones que ejecutan expresiones aritméticas. Algunas de esas expresiones simulan lo que sería un sumatorio hasta cierto número, otras realizan potencias con truncamientos, etc. Lo que se pretende simplemente es tener unas determinadas funciones que aporten cierta complejidad al ser ejecutadas.

Al igual que en el *Programa personajes* recalcábamos la gran cantidad de herencia presente, aquí la complejidad previsiblemente vendrá dada por la ejecución de las funciones inventadas. De nuevo, si el lector quiere ver con mayor detalle el código, puede ser consultado en el siguiente [enlace](#).

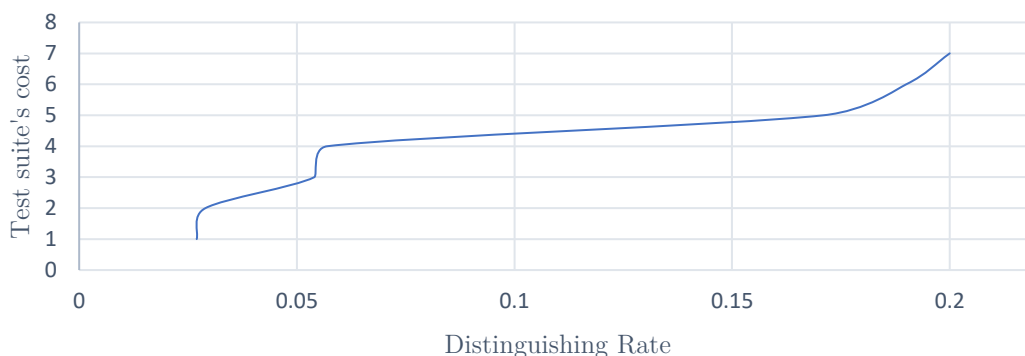


Ilustración 27: DR *Programa math* aplicando diez tests

Démonos cuenta de que aquí (aunque en menor medida) también surge el problema que vimos anteriormente sobre tests que no aportan un incremento de DR. Como veremos en la parte de análisis, el testeo de este programa es relativamente menos costoso que el *Programa personajes*, pues tras aplicar los 7 tests sólo logramos alcanzar un  $DR = 0.2$ , mientras que en el *Programa personajes*, con 7 tests el DR alcanzado fue muy cercano a 0.5. De ahí que estos “saltos” de coste que no aportan DR sean menos significativos que en el *Programa personajes*.

#### 3.4.4. Crear proyecto con programa generado

El objetivo de esta parte es generar proyectos Java de forma automática con el fin de poder medir la CT (recordemos, Complejidad de Testing) de los distintos programas y realizar comparaciones entre ellos. La parte interesante de este generador de programas<sup>10</sup> es que el usuario podrá introducir una serie de parámetros (ver **Anexo VIII**) a través de un formulario para tener un cierto control sobre la estructura que tendrá el programa generado.

De ahí que se permita analizar los datos explícitamente. El contenido de un programa generado es:

- Una constructora.
- Código generado según los parámetros introducidos por el usuario.
- Un método llamado `get_result_bool()` y otro llamado `get_result_num()`. Cada uno devolverá un array en el que se guardan los

<sup>10</sup> Es importante no confundir «Generador de programas» con «Generador de mutantes». El primero de ellos es un programa Java desarrollado por nosotros mismos que se encarga de, recibiendo una serie de parámetros, generar programas Java. El segundo es un *plugin* ya desarrollado que nosotros hemos tenido que aprender a usar e incorporar a nuestro proyecto.

resultados calculados durante toda la ejecución sobre las funciones indicadas por el usuario. Ambos métodos se encargan de las llamadas a las funciones (esto se verá en detalle más adelante).

- Los resultados de los arrays son los que se usan en el fichero que contendrá los métodos *assert*<sup>11</sup> correspondientes para aplicar el test suite, con el fin de distinguir el programa original de sus mutantes. Es por eso que el programa no sólo pide al usuario los parámetros necesarios para generar programas, sino que también pide un array de números que serán utilizados como datos de entrada del programa generado.

Para explicar el proceso que se ha llevado a cabo a la hora de diseñar el *Programa generador*, vamos a dividirlo en tres pasos y a explicar en profundidad cada uno de ellos.

### Primer paso de diseño

En primer lugar, se determinaron los tipos de las expresiones básicas necesarias. Veamos con código algunos de los ejemplos que hemos utilizado.

Condiciones de bifurcación de tipo *if*:

```
if(5 > inputs_de_usuario){
    // Cuerpo del if
}else{
    // Cuerpo del else
}
```

Bucles de tipo *for* o *while*:

```
for(int i = 0; i < 100; i++){
    // Secuencia de expresiones o más bucles anidados
}
```

---

<sup>11</sup> Los métodos *assert* se utilizan para determinar el estado de aprobación o fallo en un caso de prueba. Son proporcionados por la clase *org.junit.Assert*, que extiende la clase *java.lang.Object*. JUnit se sirve de esta clase para proporcionar un conjunto de métodos útiles para escribir casos de prueba.

Operaciones booleanas, aritméticas, asignación de variables y declaración de variables:

```
(5 - 3) > (8 * 3);
4 + 3;
int k = 8 + 2;
```

## Segundo paso de diseño

En segundo lugar, utilizamos una gramática incontextual definida por nosotros mismos (véase una breve introducción a las gramáticas incontextuales en **Anexo IX**) para definir, a un nivel más abstracto, la estructura del código de los programas generados. Veamos en detalle de qué reglas consta:

### 1. Conjunto de terminales y no terminales

#### Conjunto terminales:

```
{+, -, *, /, >, <, >=, <=, !=, numero_real,
valores_de_entrada, if, else, while, for,
condFor, (, ), '{', '}', ';', output, =,
funciones_de_assert_bool(valores_de_entrada),
funciones_de_assert_num(valores_de_entrada)}
```

#### Conjunto no terminales:

```
{Num_inputs, Programa, Programa_test,
Programa_testado, Funcion,
Funciones, Op_a, Op_r, Op_b, Exp_a, Exp_b,
Exp_simple, Expresiones, Ifs_simples, Ifs,
Fors, Whiles}
```

### 2. Estructura general del programa generado

```
Num_inputs -> valores_de_entrada
Programa -> Programa_test Programa_testado
Programa_test -> funciones_de_assert_bool(valores_de_entrada)
                funciones_de_assert_num(valores_de_entrada)
Programa_testado -> Funciones
Funcion -> BloqueExpresion Ifs Whiles Fors BloqueExpresion
Funciones -> Funcion Funciones
Funciones -> Funcion
```

### 3. Operadores

```
Op_a -> + | - | * | /
Op_r -> > | < | >= | <= | !=
Op_b -> && | or
```

#### 4. Instrucciones de asignación y expresiones simples

```
Exp_a -> numero_real Op_a Num_inputs
Exp_b -> numero_real Op_r Num_inputs
```

```
Exp_as -> Exp_as Op_a numero_real
Exp_bs -> Exp_b Op_b Exp_bs
```

```
Exp_simple -> Exp_as | Exp_bs
```

```
Expresiones -> output_a = Exp_as
Expresiones -> output_b = Exp_bs
```

```
BloqueExpresion -> Expresiones ; BloqueExpresion
BloqueExpresion -> epsilon
```

Cabe destacar que `output_a` almacenará los resultados retornados por el método `get_result_num()` y `output_b` almacenará los resultados retornados por el método `get_result_bool()`.

#### 5. Expresiones de bifurcación y bucles

```
Ifs_simple -> Expresiones
Ifs_simple -> if(Exp_bs) {Expresiones} ; Ifs_simple
Ifs -> if(Exp_bs) {Ifs} else{Expresiones | Ifs_simple}
Ifs -> Expresiones
```

```
Fors -> for(condFor) {Expresiones Fors Expresiones}
Fors -> Exp_simple
```

```
Whiles -> whiles(Exp_bs) {Expresiones whiles Expresiones}
Whiles -> Exp_simple
```

#### 6. Llamadas a funciones

Con respecto a la llamada de funciones, como hemos dicho se hace a través de `get_result_bool()` y `get_result_num()`. Ambos métodos reciben una lista de números, y cada número representa un método que se llama para ser ejecutado. Si no existe la función que se indica, se ejecutará la primera función. Después del procesamiento de todos los métodos indicados, se retornará una lista que contiene todos los resultados. Por lo tanto, el programa generador no solamente pide al usuario los atributos para generar programa, sino también una lista de valores numéricos para realizar la llamada de función.

### Tercer paso de diseño

En este último paso procedemos a definir el código Java del *Programa generador*. Como ya tenemos la gramática incontextual, lo pasamos a código Java creando e implementando

las funciones correspondientes. El mecanismo para hacer esto es bastante mecánico. Por ejemplo, las reglas de producción son  $A \rightarrow a$  y  $B \rightarrow A b$ , se hace lo siguiente:

```
public String getA(){
    String s = tratar(a);
    return s;
}

public String getB(){
    String s;
    s += getB() + tratar(b);
    return s;
}
```

El método `tratar(x)` realiza cinco funciones:

1. Evita problemas de sintaxis. Por ejemplo, añade el “;” después de cada sentencia.
2. Evita código muerto<sup>12</sup>. Por ejemplo, hace que los inputs del usuario participen en todas las expresiones básicas. De ahí que los resultados dependan directamente de los *inputs*. Por ejemplo, para la condición de bifurcación:

```
if(x > 100){
    if(y < 100){
        // Cuerpo del if
    }
}
```

El valor de las variables `x` e `y` son dos valores de entrada introducidos por el usuario. De esta forma, se garantiza que, al menos, se permite ejecutar las instrucciones del cuerpo de la condición.

3. Evita la excepción de división entre 0. Al encontrar una división, se pasa el denominador a su valor absoluto y se le suma 1.
4. Evita que los bucles *for* sean infinitos. Veamos una representación en código:

```
for(int i = x; x op1 100; x op2 1){
    // Cuerpo del for
}
```

---

<sup>12</sup> La generación de código muerto puede llegar a ser un problema grave que conviene evitar. Si existiera, las mutaciones introducidas en él generarían mutantes equivalentes al programa original (pues ahí las mutaciones no afectan en nada), y por tanto correctos. Los cálculos del DR dividen mutantes matados entre mutantes totales, pero esa división es válida sólo si todos los mutantes (o casi todos) son realmente incorrectos. No podemos permitirnos analizar cada mutante para ver si es correcto (eso es indecible en general). Por tanto, necesitamos que todos, o la inmensa mayoría de los mutantes, sean incorrectos. Eso requiere que no tengan código muerto donde introducir mutaciones “inocuas”.

Mientras se genera el código, se verifica la relación entre la variable **i** y los operadores **op1** y **op2**. O sea, como vemos a continuación, básicamente se usan dos reglas.

```
si x < 100 entonces op1 = {<} and op2 = {++}
si x > 100 entonces op1 = {>} and op2 = {--}
```

5. Evita que los bucles *while* sean infinitos. Cada vez que se genera una expresión *while*, se crea una variable **x** de tipo entero que se va modificando con cada iteración del bucle. Con respecto al valor que se le asigna a dicha variable y el operador al que se le aplica, se hace lo siguiente:

```
int x;
while(x op1 num_iteracion){
    // Cuerpo del while
    x op2;
}
```

Suponiendo que **num\_iteracion** es un valor conocido, establecemos una relación entre **x**, **k**, **op1** y **op2** de la forma:

```
si op1 es < entonces x = 0, op2 es operador de suma a 1
si op1 es > entonces x = num_iteracion * 2, op2 es operador de resta a 1
```

Una vez hemos entendido cómo se realiza la generación de código, estamos en situación de explicar las limitaciones que pueden surgir cuando tratamos de generar un programa de forma automática (si el lector tiene interés por conocer el código completo del programa utilizado para generar programas, puede consultarlo en el siguiente [enlace](#)).

### Limitaciones a la generalidad del código de los programas generados

Cuando delegamos alguna tarea manual a un programa para que éste la ejecute automáticamente por nosotros, debemos adelantarnos a todos los problemas que pudieran surgir cuando se lleva a cabo la generación automática de programas. En este sentido, hemos introducido una serie de limitaciones a la generalidad del código generado que, si no se introdujeran, probablemente generarían programas incorrectos con instrucciones que nunca se ejecutarán, errores en tiempo de compilación, programas con bucles infinitos, código muerto, etc. Recordemos que utilizaremos nuestro generador de programas para generar el programa correcto cuyos mutantes testaremos posteriormente, por lo que debemos garantizar que los programas generados por nuestro generador eviten los problemas anteriores, y así puedan considerarse correctos. A continuación, mostramos cuáles son esas limitaciones a la generalidad del código de los programas generados.

- Dentro de un tipo de bucle no aparecerá otro tipo de bucle. No puede darse la siguiente situación:

```
while(condición){
    for(condición){
        // Cuerpo del for
    }
}
```

- El valor de los operandos en las expresiones es aleatorio y está establecido entre 1 y 100.
- En el caso de que tuviéramos una división cuyo denominador fuese 0, siempre se le calcula su valor absoluto y se le suma 1.
- La condición de cada bucle *for* no depende de ningún valor de entrada proporcionado por el usuario, por eso siempre se garantiza que finalizará.
- Toda condición *if* tiene su *else* correspondiente. Nunca aparecerá un *if* solo. El siguiente ejemplo nunca podría suceder.

```
if(condición){
    if(condición){
        // Cuerpo del if
    }
}else{
    // Cuerpo del else
}
```

- Para evitar código muerto, optamos por la opción de que no haya asignación de variable a otra variable. En otras palabras: a las variables sólo se les asignan valores primitivos.

Con respecto a la generalidad de los programas generados automáticamente, consideramos que son lo suficientemente generales como para permitirnos realizar nuestros experimentos de testing con ellos a pesar de las limitaciones a la generalidad que hemos impuesto, pues contienen suficientes instrucciones (bucles, expresiones aritméticas, lógicas, asignaciones, llamadas a función, etc).

### Programas de testing generados

Dado que es necesario comparar los inputs y outputs entre los mutantes y los resultados esperados, el generador también genera un programa de test para testear el programa que se acaba de generar. Además del código necesario para que se ejecute el test, se añaden dos reglas. Una es para comparar resultados de tipo booleano y otra para comparar resultados de tipo numérico. La regla tiene la siguiente sintaxis:

```
assertArray(resultado esperado, resultados de mutantes, tolerancia error);
```



Por lo tanto, al aplicar el test, se compara el resultado esperado con el resultado de los mutantes para decidir si el estado de cada mutante es *killed* o *lived*.

### 3.4.5. Preprocesado

Partamos de que el usuario ya ha ejecutado un proyecto Java (e.d. o bien se ha cargado un *Programa real* o bien se ha generado un *Programa generado*, tal y como se observa en la **Ilustración 23**), por lo tanto, en este punto tenemos los ficheros de las clases que se van a mutar y los tests que se van a testear en el servidor. El proceso que se lleva a cabo en este momento es el de «Preprocesado», proceso que hará posible que la herramienta PIT pueda generar todos los mutantes para cada una de las clases Java y, posteriormente, obtener resultados al aplicar las pruebas. Este proceso se lleva a cabo gracias al script *preprocesar.sh*. Entrando más en detalle, las tareas realizadas son las siguientes:

1. Se limpian los directorios del proyecto PIT<sup>13</sup> para evitar conflictos con programas anteriores.
2. Se mueven los ficheros al proyecto PIT.
3. Se configura el fichero de configuración con los tipos de mutaciones que se van a utilizar en PIT.
4. Se configura el fichero de configuración de PIT con las clases Java.
5. Se configura el fichero de configuración de PIT con los tests *JUnit*.

Ya que este proceso es algo tedioso y que se ejecuta de manera totalmente *invisible* de cara al usuario, consideramos que debe estar fuera de las explicaciones de esta sección de la memoria. En cualquier caso, puede consultarse el código y los pasos con más detalle en el **Anexo X**. Al mismo tiempo, recomendamos consultar el **Anexo XI** para conocer los comandos Unix que se han utilizado.

### 3.4.6. Base de datos

En cuanto a la tarea de almacenar datos, se ha utilizado el sistema de gestión de bases de datos MariaDB, que viene con el paquete XAMPP. La base de datos, de nombre *dbtfg*, está compuesta por tres tablas donde se guarda la información de los proyectos Java (tabla *proyectos*), los tests que se aplican a un determinado proyecto Java (tabla *test\_proyecto*) y los resultados de aplicar un test a los mutantes generados (tabla *mutante\_test\_proyecto*).

---

<sup>13</sup> Decimos “proyecto PIT” a aquel proyecto Java ejecutado mediante PIT a través del cual, mediante comandos, llevamos a cabo la generación de mutantes. Es decir: cuando se ha cargado un proyecto Java desde la interfaz, éste pasa a ser ejecutado por PIT para generar mutantes. A partir de ese momento es más correcto hablar de “proyecto PIT” que de “proyecto Java”.

Veamos una ilustración donde se presentan estas tres tablas, sus relaciones y atributos. Para ver con más detalle las características de las tablas y de la base de datos utilizada, puede consultarse el **Anexo XII**.

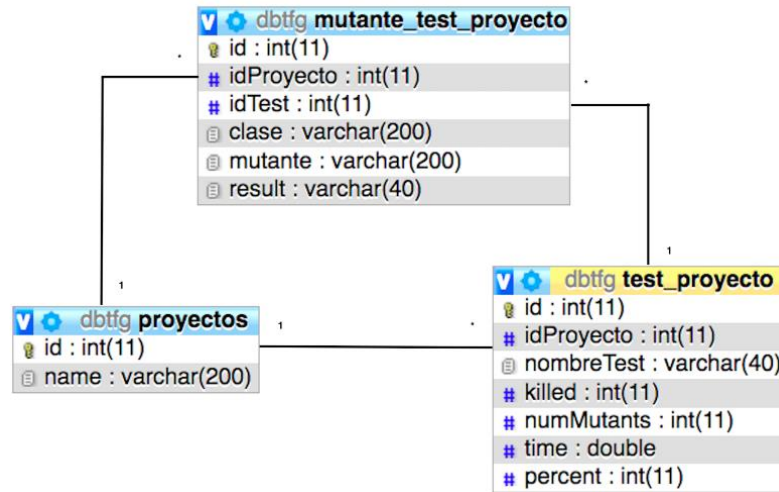


Ilustración 28: Esquema de la base de datos

## 4. Análisis de experimentos

Pasamos a abordar los análisis sobre los experimentos realizados en las dos fases del TFG. Durante la primera fase nos centraremos en analizar la CT (recordemos, complejidad de testing) para los tres ejemplos del artículo *Introducing Complexity to Testing* simulados empíricamente. Ya en la segunda fase generaremos, de manera automática, numerosos programas (variando la estructura de los mismos) de forma que nos permitan analizar y obtener algunas conclusiones.

Nos gustaría mencionar un detalle muy importante a la hora de hablar de complejidades (sobre todo referente a la primera Fase I). Cuando en el artículo se prevén de manera teórica las complejidades de los tres ejemplos tratados en nuestra Fase I, dicha complejidad se obtiene en el límite. En nuestro caso, las gráficas han sido creadas basándonos en un conjunto finito de experimentos, luego la complejidad observada está acotada. Si, por ejemplo, observamos el primer ejemplo artículo, vemos que la CT calculada se plantea como  $O(\log(n) \cdot \log(\log(n)))$ . Eso quiere decir que la complejidad teórica está acotada linealmente por una función  $k1 \cdot \log(n) \cdot \log(\log(n)) + k2$ , donde el coeficiente y término independiente son desconocidos. Esto permite que cualquier conjunto finito de puntos observados pueda ser consistente con cualquier complejidad  $O(\log(n) \cdot \log(\log(n)))$ , por lo que no será posible garantizar que nuestras observaciones procedan de funciones que respeten las complejidades teóricas previstas. Por ello, al comparar las formas de las funciones previstas y de las gráficas observadas, sólo nos limitaremos a considerar plausible que unas sean consistentes con respecto a las otras.

### 4.1. Análisis de la Fase I

Para llevar a cabo el análisis de la primera fase vamos a realizar una distinción basándonos en la CT obtenida en los ejemplos del artículo. Ya que el primer y segundo ejemplo del artículo tienen la misma CT asintótica, realizaremos sus análisis al mismo tiempo. Posteriormente, pasaremos a analizar el tercer ejemplo del artículo.

Para realizar un análisis de complejidad es necesario que calculemos la CT basándonos en el DR de las gráficas que vimos durante la simulación de los ejemplos. En el eje  $x$  se muestra la DR objetivo denotada en una cierta escala especial, y en el eje  $y$ , el coste que debe tener un test suite completo para alcanzar dicha DR. Concretamente, el eje  $x$  muestra el resultado de la expresión  $1/(1 - DR)$ , que es lo que hemos denotado con el nombre de *Alpha* (e.g. si queremos obtener un  $DR = 0.75$ , entonces tendremos que alcanzar un *Alpha* = 4, pues despejando DR de la fórmula anterior, obtenemos que  $1 - 1/Alpha = DR$ ). Las complejidades calculadas en el artículo se definen en estos términos, por lo que cualquier comparación requerirá que nuestras complejidades observadas lo hagan también.

Cabe recordar que los test suites aplicados consisten en probar todas las combinaciones posibles de *inputs* hasta una determinada longitud (e.d. hasta un determinado nivel en el árbol de ejecución), asumiendo que hay sólo dos *inputs* posibles (e.g. **a** y **b**).

#### 4.1.1. Análisis de los dos primeros ejemplos simulados

Después de aplicar diferentes test suites, incrementando su profundidad hasta nivel 6 a los 100.000 mutantes, obtenemos la siguiente gráfica de CT para el primer ejemplo del artículo (recordemos que, para nosotros, el nodo raíz de los árboles es el nivel 1 y que se comienza a testear desde los arcos que llevan a sus hijos, es decir, al nivel 2):

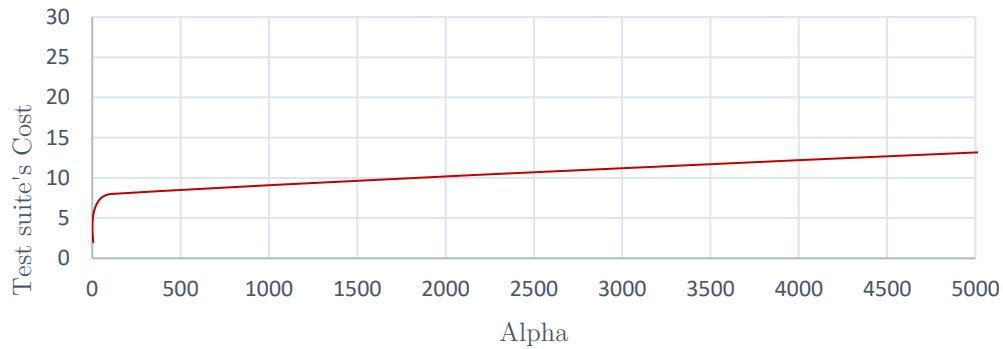


Ilustración 29: CT en primer ejemplo con test suite

Y para el segundo ejemplo del artículo hemos obtenido la siguiente gráfica:

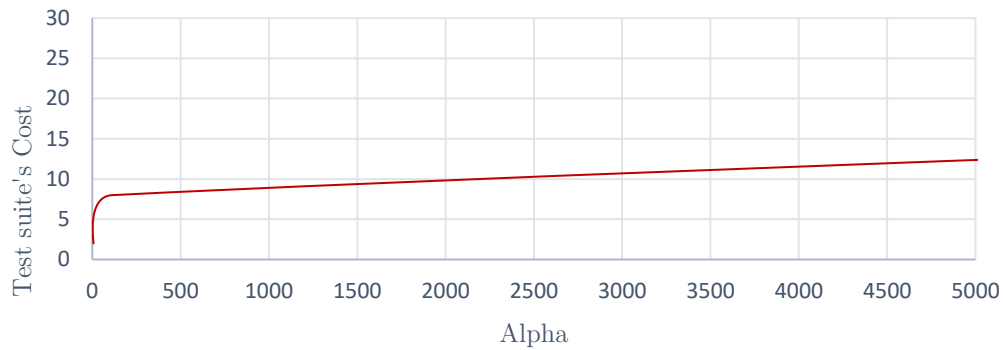


Ilustración 30: CT en segundo ejemplo con test suite

Tal y como se mencionó durante nuestra explicación de cada uno de los dos ejemplos del artículo, la complejidad prevista en el artículo cuando se considera que el coste es el número de *inputs* aplicadas entre todos los tests del test suite aplicado es  $O(\log(n) \cdot \log(\log(n)))$  (e.d. como en nuestros experimentos) en ambos ejemplos.

Es importante recordar que, debido a los medios limitados de los que disponemos, tan sólo hemos logrado obtener un  $DR < 1$  para los cuatro primeros niveles de los árboles (en particular, considerando mutantes con errores hasta nivel 6). En otras palabras, cuando se realiza un test suite de nivel 5 o superior, ya se alcanza un  $DR = 1$ . Nótese que en realidad

esto no es cierto para un número arbitrariamente grande de mutantes y de mucha más profundidad respecto a la nuestra, pues simplemente se irían obteniendo DRs cada vez más cercanos a 1 pero sin llegar a él.

Dicho esto, observemos en las dos ilustraciones anteriores cómo al principio crece relativamente rápida la CT al aplicar un test suite de nivel 3 pero, a medida que se profundizan niveles, la complejidad tiende a crecer relativamente más despacio. Pero ¿cómo es la CT que representan esas gráficas? ¿Tienden a ser del orden de las CT teóricas que aparecen en el artículo? Para realizar un análisis lo más eficaz posible se ha generado una gráfica del orden de complejidad previsto en el artículo y, para las dos gráficas de CT obtenidas, hemos hecho la comparación.

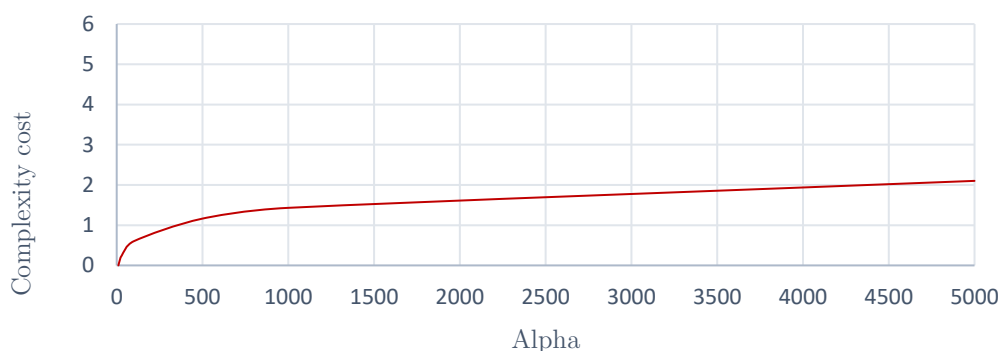


Ilustración 31: Función descrita por  $\log(n) \cdot \log(\log(n))$

Observamos que la gráfica descrita por la función  $\log(n) \cdot \log(\log(n))$  ciertamente tiende a ser parecida a la de los dos primeros ejemplos del artículo. No obstante, no es posible tener certeza total en la coincidencia entre las complejidades observadas experimentalmente y la teórica prevista, como se dijo anteriormente. En particular, la comparación entre ambas, incluso una informal, se encuentra con los siguientes problemas:

- a) Limitaciones en los experimentos realizados. Debido a nuestros medios limitados, tan sólo somos capaces de realizar una aproximación a la complejidad mencionada en el artículo. A medida que generamos una mayor cantidad de mutantes y de mayor profundidad (e.d. realizamos experimentos con un mayor alcance), la CT tiende a ajustarse y parecerse más a la que estamos persiguiendo.
- b) Lo que conlleva el problema anterior es que tan sólo estamos viendo una pequeñísima parte de lo que sería un número arbitrariamente grande de experimentos (es lo que mencionábamos anteriormente sobre considerar “tan sólo” un número finito de experimentos). En otras palabras: tan sólo estamos viendo una pequeña parte de la “foto” que, si bien es suficiente para ver que las complejidades tienden a parecerse, no lo es para hacer una afirmación absoluta de que esto será así cuando  $n$  es un número arbitrariamente grande (tanto en número de mutantes como en niveles de profundidad).

Aunque no podemos asegurar que la CT cuando  $n$  continúe creciendo sea del orden de  $O(\log(n) \cdot \log(\log(n)))$ , sí que podemos afirmar que, en vista de las observaciones realizadas, resulta intuitivamente razonable que las CT de los dos primeros ejemplos del artículo simulados con nuestro programa Java tiendan a ser de dicha complejidad.

#### 4.1.2. Análisis del tercer ejemplo simulado

Análogamente, procedemos a realizar el análisis del tercer ejemplo simulado. Al igual que en los dos casos anteriores, las pruebas realizadas han sido sobre 100.000 mutantes, sólo que esta vez, como dijimos, los mutantes tienen profundidad 7 (recordemos que, para nosotros, la raíz del árbol es el nivel 1). Dado que en este caso el DR obtenido es significativamente menor con diferentes test suites hasta alcanzar aquellos de profundidad 6 (como vimos en la **Ilustración 20**), la CT será considerablemente mayor respecto a los dos primeros ejemplos.

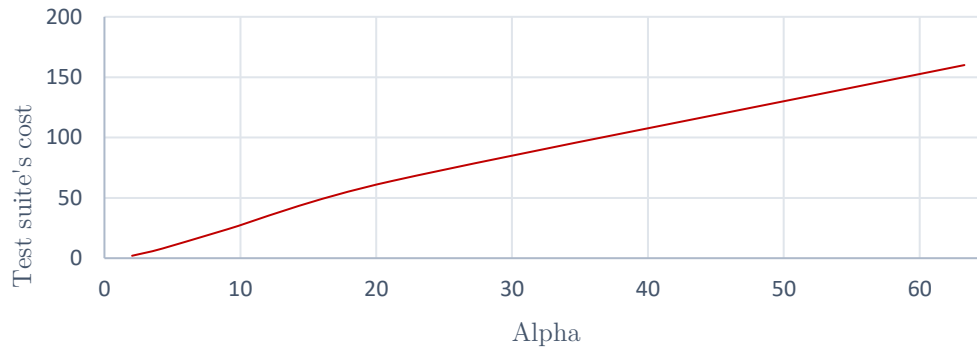


Ilustración 32: CT en tercer ejemplo con test suite

En efecto, si observamos la gráfica vemos cómo la CT aumenta con mayor rapidez respecto a los dos primeros ejemplos. Esto es debido a que en este ejemplo estamos asumiendo una estrategia en la que el comportamiento de la IUT tiene como mucho un error, así que encontrar ese error es muchísimo más difícil respecto a los dos primeros ejemplos.

En este tercer ejemplo vimos que, de forma general, la complejidad cuando el coste al aplicar un test suite es el número total de *inputs* es del orden de  $O(n^{1/r} \cdot \log(n^{1/r}))$ . Para considerar un caso concreto en nuestros experimentos, en cálculos hemos tomado  $r = 1$ , luego resolviendo la división del exponente obtenemos que la CT es del orden de  $O(n \cdot \log(n))$ . De nuevo, adjuntamos la gráfica para tener una idea de cómo es la función descrita por  $O(n \cdot \log(n))$ .

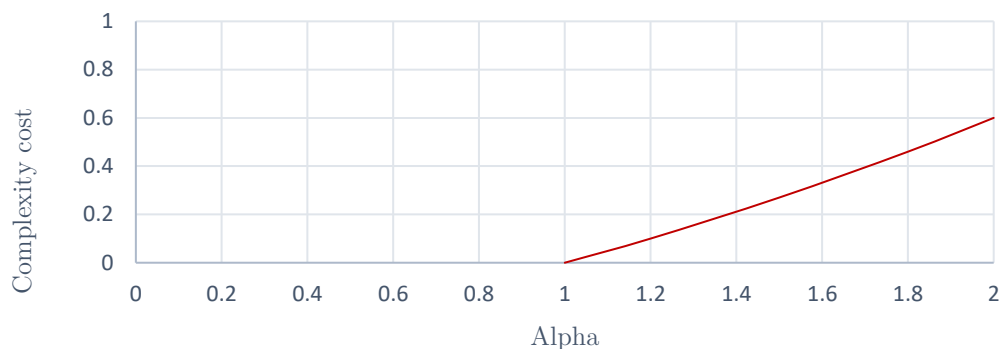


Ilustración 33: Función descrita por  $O(n \cdot \log(n))$

De nuevo, si bien es cierto que la gráfica descrita por la función  $O(n \cdot \log(n))$  ciertamente tiende a ser parecida a la de los experimentos realizados para la simulación de este tercer ejemplo, no podemos afirmar con certeza total la coincidencia de ambas complejidades. Las dos principales razones son exactamente las mismas que las explicadas durante los experimentos de los dos primeros ejemplos.

## 4.2. Análisis de la Fase II

Antes de comenzar con los análisis de la segunda fase del TFG, debemos abordar una cuestión muy importante respecto al testing aplicado a los proyectos Java. Al igual que ocurre en la parte de *Programa generados*, es posible que un usuario cree un programa y lo importe para aplicar pruebas. Dicho programa tendrá una forma u otra, y dicha forma podrá clasificarse en términos de los mismos parámetros que usamos para generar nuestros programas con el programa generador (e.d. cierto número de iteraciones, de anidamiento, de condiciones lógicas, etc).

Al utilizar nuestro generador de programas para generar un determinado programa con unos determinados parámetros de generación (número de *ifs*, etc.), obtenemos un programa al azar que cumple dichos parámetros. Imaginemos que deseamos testear el programa obtenido con un determinado test suite consistente en un único test fijado por nosotros. Debido al azar en la generación del programa, cabe preguntarse si el DR obtenido será muy sensible al programa concreto obtenido. Hemos realizado un experimento para medir dicha sensibilidad.

Observamos en la siguiente tabla un experimento que consiste en generar sesenta programas de la siguiente forma: se generan los diez primeros poniendo todos sus parámetros de generación (*ifs*, etc.) a valor 1. A continuación, se generan otros diez poniendo todos sus parámetros a valor 2. Y así sucesivamente hasta llegar a parámetros con valor 6. Para los sesenta programas generados, se adjunta el DR de cada uno de ellos tras aplicar un test suite con un único test fijo (las mismas pruebas en cada uno de ellos), la media del DR obtenido en cada serie de diez programas generados con los mismos parámetros, la varianza y el porcentaje que supone dicha varianza respecto de la media. Como vemos, el porcentaje

en el que el DR varía tomando muestras de diez programas para cada serie es prácticamente despreciable.

	Default = 1	Default = 2	Default = 3	Default = 4	Default = 5	Default = 6
<b>Distinguishing Rate</b>	0.135	0.095	0.079	0.118	0.066	0.084
	0.129	0.06	0.091	0.073	0.087	0.097
	0.101	0.08	0.065	0.072	0.057	0.131
	0.105	0.078	0.103	0.091	0.063	0.095
	0.14	0.092	0.099	0.081	0.079	0.098
	0.139	0.086	0.079	0.089	0.118	0.113
	0.115	0.063	0.095	0.08	0.091	0.059
	0.154	0.092	0.09	0.125	0.083	0.076
	0.154	0.112	0.075	0.072	0.094	0.099
	0.113	0.086	0.063	0.105	0.079	0.088
<b>Media</b>	0.1285	0.0844	0.0839	0.0906	0.0817	0.094
<b>Varianza</b>	0.00033165	0.00021084	0.00017249	0.00033504	0.00028061	0.0003466
<b>Porcentaje</b>	4.2617E-07	1.77949E-07	1.44719E-07	3.03546E-07	2.29258E-07	3.25804E-07

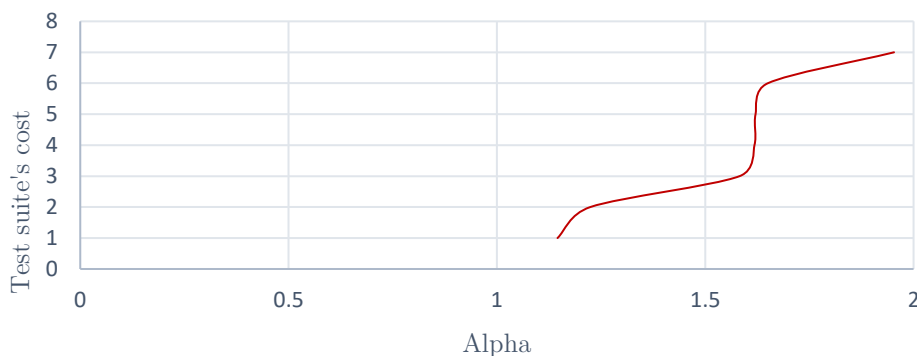
Tabla 1: Varianza del DR en programas con parámetros de generación diferentes

En vista de los resultados obtenidos, durante los siguientes apartados la posible variación de DR entre diferentes programas generados con los mismos parámetros de generación no se tendrá en cuenta, por lo que no generaremos varios programas con los mismos parámetros para estudiar la variabilidad de la DR en ellos.

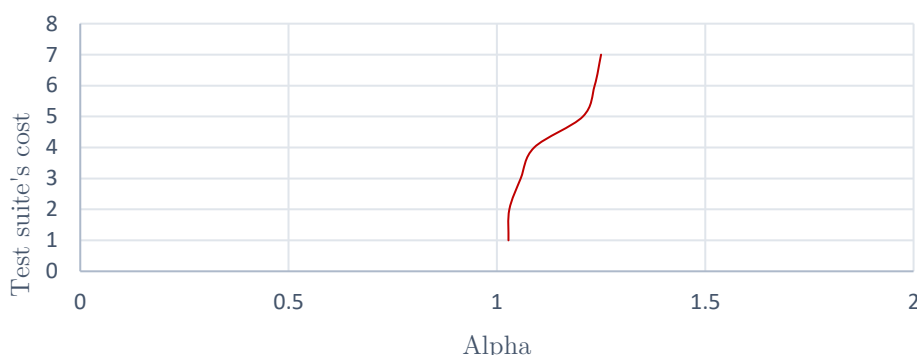
#### 4.2.1. Análisis de programas reales

Vamos a proceder a analizar las CT de los dos programas que desarrollamos manualmente. Recordemos que en esta sección los tests incluidos en los test suites no han sido escogido al azar, sino que han sido diseñados a mano para tratar (como vimos en las gráficas de DR esto no fue posible en todos los casos) de alcanzar cada vez una mayor cobertura. En este caso, creemos más conveniente presentar primero las gráficas y después proceder a analizarlas. En primer lugar, observemos la gráfica de CT obtenida para el *Programa personajes*.



Ilustración 34: CT en *Programa personajes*

Y, en segundo lugar, observemos la gráfica obtenida para el *Programa math*.

Ilustración 35: CT en *Programa math*

A pesar de que obviamente los test suites que se han aplicado en ambos programas son distintos, el coste de los test suites es calculado de la misma manera en los dos casos. En particular, el coste viene dado por el tamaño de los test suites que se van aplicando, es decir: se comienza aplicando un test de coste de coste 1. A continuación se aplica un test de mayor alcance (que incluye el test anterior) de coste 2. Y así sucesivamente hasta llegar a los 7 tests aplicados.

Fijémonos también en que, tras aplicar los 7 tests, el valor de *Alpha* alcanzado (y el correspondiente valor de DR que representa) en cada programa es distinto: el *Programa personaje*, tras aplicar sus diez tests, llega a casi las 2.000 unidades, mientras que el *Programa math*, tras aplicar sus diez tests, tan sólo llega a las 1.250 unidades. Por lo tanto, la función que describe la CT del *Programa math* crece considerablemente más rápido.

Nótese que algunos tests no aportan prácticamente ninguna mejora en el testeo del programa (esto es algo que ya se vio en las gráficas de DR), por eso aparecen pequeños tramos entre tests en los que la complejidad crece verticalmente. Considerando que los tests se han diseñado manualmente, esto también es lo esperable, pues para que un test siempre consiga encontrar errores que los tests anteriores no pudieron encontrar, dicho test

tendría que diseñarse haciendo un estudio exhaustivo sobre las líneas de código que los anteriores tests no pudieron cubrir, por qué no las pudieron cubrir y qué podría hacerse para que sean cubierto (e.d. debería mejorarse la estrategia de testing a seguir). Estudio que, de nuevo, no se ha llevado a cabo debido a la falta de tiempo y el esfuerzo que supondría. Pero lo que sí se ha hecho es diseñarlos asegurando que cubren casi el 100% de cobertura de código, tal y cómo se mencionó en el desarrollo.

Para terminar con el análisis, nos gustaría destacar cómo ha crecido la CT sobre los dos programas. Recordemos que el *Programa personajes* tenía un mayor número de clases y ciertamente poseía una mayor herencia que el *Programa math*, pero este segundo estaba formado por algunos métodos que, en la mayoría de los casos, tenían muchas más condiciones de bifurcación, iteraciones en bucles, condiciones lógicas, etc. Luego los análisis efectuados corroboran que testear programas cuya estructura sea más parecida al *Programa math* es considerablemente más costoso que testear programas cuya estructura se parezca al *Programa personajes*.

#### 4.2.2. Análisis de programas generados

Después de realizar un análisis con programas que trataban de simular lo que sería una programación hecha a mano, pasamos a un escenario completamente distinto en el que vamos a estudiar qué elementos de los programas (e.d. número de condiciones, operaciones aritméticas, bucles, etc.) tienen una mayor influencia en la CT observada cuando se aplican pruebas.

A continuación vamos a realizar un análisis en dos escenarios que tienen una pequeña diferencia entre sí. Como dijimos cuando se explicó el desarrollo del programa generador (ver 3.4.4), éste recibe una serie de parámetros (atributos del programa) a través de un formulario HTML. Las pruebas que se han ejecutado son:

- a) *Variación en un solo atributo*: se procederá a generar diferentes programas variando uno de los atributos que recibe el generador de programas, y manteniendo los demás parámetros a los valores mínimos por defecto. Así, por ejemplo, un programa podría comenzar teniendo una sola anidación *if*. A continuación, se genera otro programa que tenga dos anidaciones *if*. Luego, tres anidaciones, etc., y todo ello mientras los demás parámetros se mantienen fijos. Hacemos lo mismo con cualquiera de los atributos disponibles que podemos modificar (y que veremos en detalle en la siguiente ilustración). El objetivo concreto de variar un solo atributo es ver qué elementos disminuyen más el DR alcanzado a medida que se van incrementando. En otras palabras, un programador podría hacerse la siguiente pregunta: “¿Qué estructura de programa será más difícil de testear, e.g. uno en el que aparezcan muchas condiciones *if* o uno que posea muchos bucles *while*?”

- b) *Variación en dos atributos*: una vez obtenidos resultados de DR para la variación de un atributo, estamos interesados en analizar qué ocurre cuando los atributos que van variando lo hacen de dos en dos. En este segundo caso, como veremos, contestaremos a pregunta del tipo: “¿Si juntamos un atributo cuyo impacto sobre el DR sea bajo con otro cuyo impacto sea alto, de alguna forma se equilibrará el DR total? ¿Cómo de drástica será la caída del DR cuando aumentamos dos atributos cuyo impacto sobre el DR obtenido en el caso (a) sea bajo?” En definitiva, testaremos diferentes combinaciones de atributos y veremos cómo crece (o decrece) el DR a medida que aumentamos el valor de los atributos.

Cabe destacar que, mientras se está realizando el incremento de un atributo, el resto de los atributos toman su valor mínimo. Por ejemplo: si se está incrementando la anidación de *if*, entonces todos los demás atributos tendrán un valor de 1 (e.d. habrá un solo bucle *while* con una sola iteración, también habrá una sola condición lógica, etc.). Además, para ambas pruebas (a) y (b) se han aplicado los tests de manera que, cada vez que uno es aplicado, se incrementa la *longitud* (el alcance del test). De esta forma, cada una de las pruebas testea, además, el “camino” que testeó su prueba antecesora, tal y cómo ocurriría al aplicar un conjunto de tests de distinta longitud a un árbol binario de ejecución, solo que esta vez lo que se aplica es un segundo test (e.d. un test suite de tamaño 2).

En la siguiente gráfica se han generado un total de 90 programas (número de barras en la gráfica). Siendo más específicos, se han generado 10 programas para cada tipo de parámetro. Para entender mejor esta idea, tomemos como ejemplo el primer parámetro que aparece en la gráfica (**anidacionIf**, en color naranja oscuro). La primera barra que aparece corresponde a la generación de un programa Java cuyo parámetro es **anidacionIf = 2** (dejando el resto de los parámetros con valor 1). La segunda barra corresponde a un valor **anidacionIf = 3**, y así sucesivamente hasta llegar a **anidacionIf = 11**. Como vemos, el DR decrece a medida que el programa contiene un mayor número de anidaciones *if* (tal y como se espera que sea). La siguiente barra de la gráfica (color naranja algo más claro) ya pertenecería a un programa generado cuyo parámetro **anidacionIf** pasaría a valer 1 (pues ya no se tiene en cuenta) y ahora, el parámetro a incrementar sería **anidacionWhile** empezando desde el valor 2, y así. Análogamente, se hace lo mismo para el resto de los parámetros. Dicho esto, vamos a ver qué gráfica hemos obtenido.

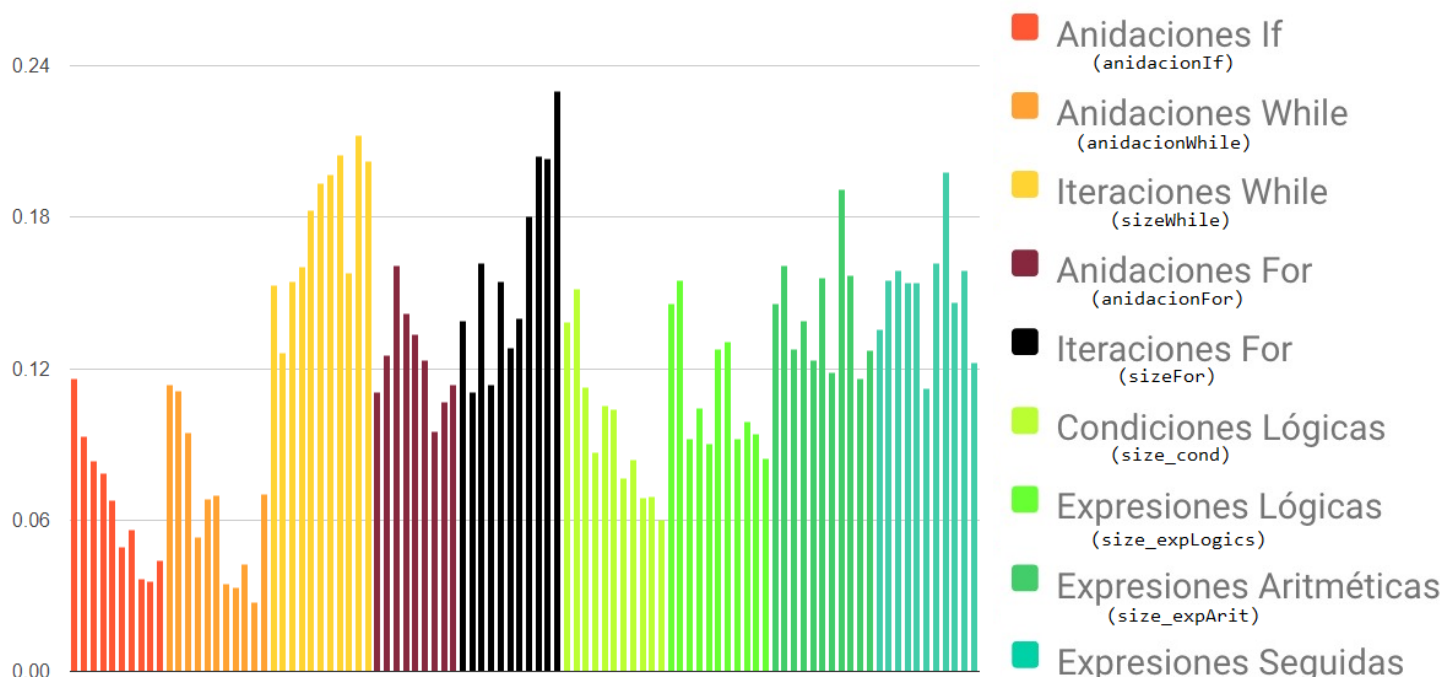


Ilustración 36: DR obtenido incrementando un solo atributo

Viendo los resultados de DR obtenidos podemos observar algunas diferencias notables. Fijémonos en cómo, a medida que incrementamos la anidación en *if* y *while*, el DR que obtenemos cae considerablemente. Y que lo mismo ocurre cuando incrementamos el tamaño de las condiciones lógicas. Por el contrario, en color amarillo y negro vemos cómo, a medida que se incrementa el número de iteraciones *while* y *for*, el DR aumenta. Esto es lo que se espera, pues a medida que vamos iterando en un bucle, las variables toman valores diferentes, lo que permite realizar más distinciones entre el programa original y los mutantes. Nótese también que incrementar el número de expresiones lógicas (que no el tamaño de las mismas), expresiones aritméticas o expresiones seguidas, no aporta ningún beneficio de DR, sino que más bien tiene un efecto aleatorio.

Por último, cabe destacar que el incremento de anidaciones *for* no hace que disminuya el DR. Esto se debe a que la condición para que un *for* ejecute sus iteraciones no depende del valor que el usuario introduzca como parámetro (debido a la forma en la que el *Programa generador* construye los programas).

Yendo a nuestro segundo escenario, en el que los atributos se incrementan de dos en dos, nos encontramos con una gráfica en la que se reflejan algunas de las consecuencias esperadas. Veamos cómo, al incrementar dos atributos que por sí solos hacen que se alcance un DR relativamente bajo, el resultado es significativamente peor. Tomemos la siguiente muestra y analicémosla.

En la primera gráfica vemos que el DR obtenido para anidaciones *if* es cercano al 0.12, y que el DR obtenido para anidaciones *while* es también muy cercano a 0.12. Sin embargo,

al observar la primera prueba de la segunda gráfica (que se adjunta a continuación), vemos cómo el DR ni siquiera alcanza un valor cercano a 0.12, y que tan sólo llega a 0.09 (diferencia que no es muy notable cuando hablamos en un intervalo  $[0,1]$ , pero sí que lo es en nuestro caso, pues todas las pruebas realizadas, como puede verse, ni siquiera llegan a un  $DR = 0.25$ ). Por lo general, esto ocurre para todo par de atributos para los que en la primera gráfica fue relativamente difícil obtener un DR alto.

También podemos observar como en **Anidaciones For** y **Anidaciones If** el DR obtenido tiende a ser crecer más rápido respecto al atributo **Anidaciones If** visto en la gráfica anterior. Deducimos pues que el número de anidaciones *for* (y la variación de sus variables a medida que se van recorriendo dichas anidaciones) son propensas a cubrir con más facilidad ciertas líneas de código, por lo tanto, un mayor número de anidaciones *if*. Esto hace que, de nuevo, podamos hacer una mayor distinción entre el programa original y los mutantes generados.

Por último, fijémonos en los primeros siete experimentos de la segunda gráfica. Se observa que, cuando hay código que se introduce dentro de anidaciones *if*, éste tiende a ser más difícil de distinguir (e.d. es más difícil detectar errores y se obtiene una DR peor). De ahí que, a medida que aumentamos el número de anidaciones *if* (aunque dentro haya cualquier otro atributo), el DR caiga rápidamente (llegando incluso a valores relativamente bajos, cercanos a 0.03 en la última prueba para cada uno de los siete experimentos).

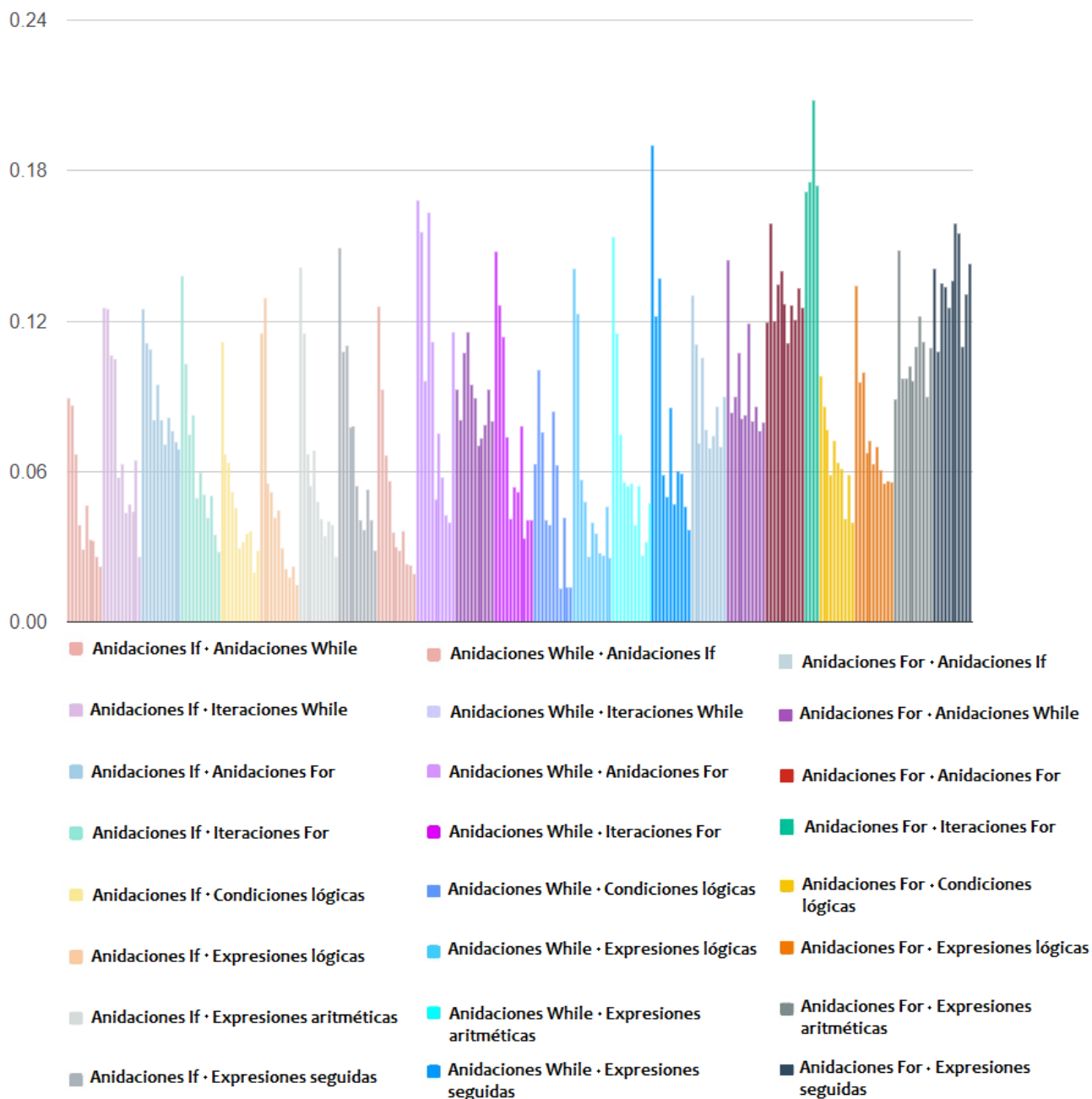


Ilustración 37: DR obtenido incrementando dos atributos

#### 4.2.3. Análisis del impacto en los tipos de mutaciones

Hasta ahora, durante la segunda fase hemos analizado qué estructuras en los programas son relativamente más complejas de testear. En esta última sección de los análisis vamos a estudiar qué tipo de mutaciones son más fácilmente detectables cuando se introducen como errores. Aprovechando la potencia que ofrece Pitest a la hora de introducir mutaciones en el código (recordemos que Pitest tan sólo introduce una sola mutación en cada mutante), pasamos a analizar qué tipo de mutaciones se detectan con mayor facilidad. En este sentido, un programador podría estar interesado en saber, por ejemplo, si un error en el incremento de una variable será más fácilmente detectable respecto a un error en una condición lógica.

Observemos la gráfica que se adjunta a continuación. En primer lugar, para las pruebas se han aplicado los mismos tests que en los dos análisis anteriores. Para cada tipo de mutación posible, hemos realizado varios experimentos en los que dicho tipo de mutación es la única introducida en los mutantes generados. En cada tipo de mutación considerada hemos generado programas de forma que hemos incrementado sucesivamente todos los atributos (parámetros que recibe). Por ejemplo, **Default = 2** indica que se ha generado un programa cuyos parámetros (todos ellos) tienen un valor igual a 2. Así, se han ido generando programas hasta llegar a **Default = 6**. Por otro lado, observamos que en el eje **x** aparecen todos los tipos de mutaciones que permite Pitest (recordamos que pueden consultarse en el **Anexo II**). Además, hemos incluido algunos subtipos de mutaciones (como **RemoveConditionalMutator\_EQUAL\_ELSE** y sus variantes).

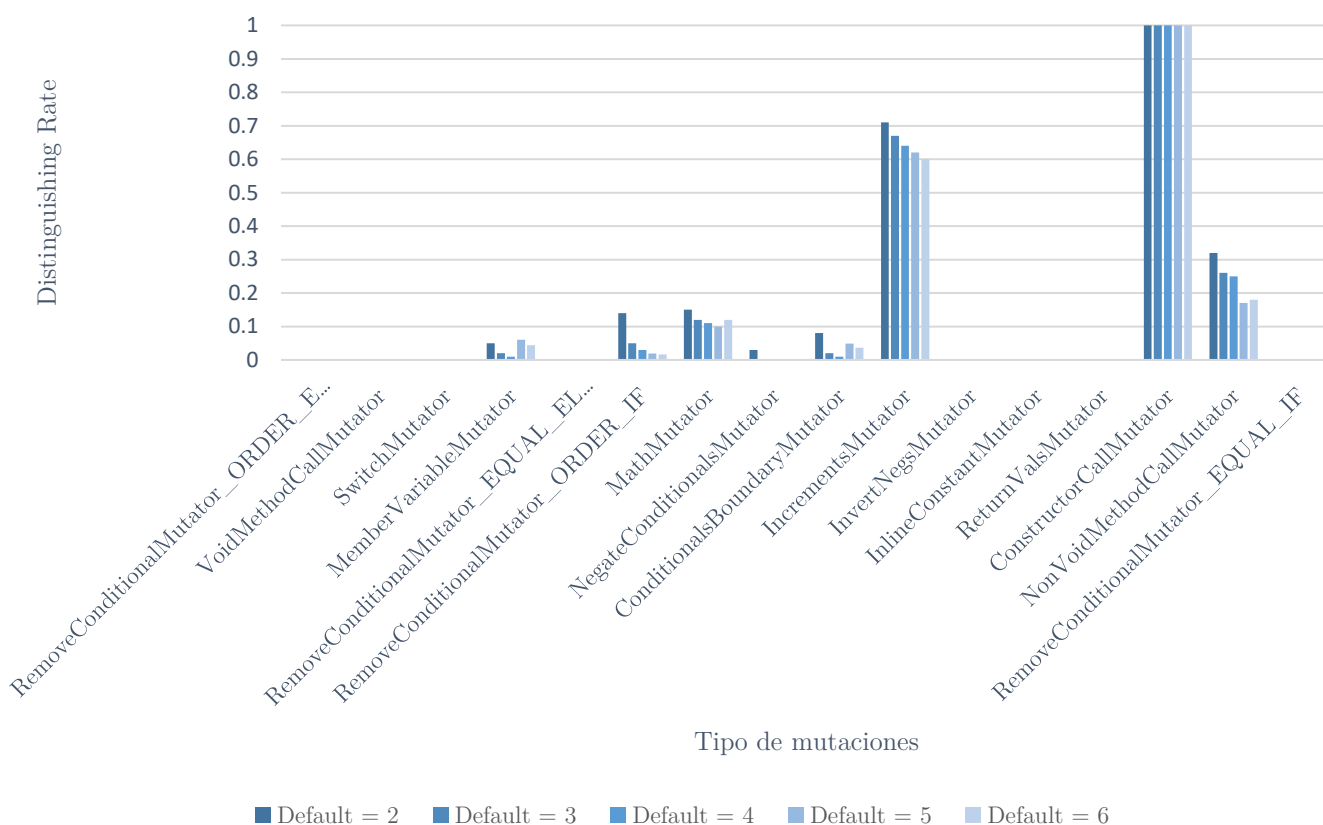


Ilustración 38: DR según el tipo de mutación

Cabe destacar que los programas generados automáticamente no poseen una *riqueza* total en lo que a código se refiere (e.d. no aparecen todas las instrucciones sobre las que Pitest podría hacer mutaciones). La consecuencia es que hay determinadas mutaciones que no llegan a introducir errores en el código porque el código que debería mutarse ni siquiera existe en los programas generados. Viendo la gráfica, detectamos que, por ejemplo, el tipo de mutación **SwitchMutator** no llega a generar ningún mutante (pues, repetimos, no encuentra código que mutar). Respecto al resto de tipos de mutaciones en los que ocurre

lo mismo, no podemos hacer ninguna afirmación sobre si son más o menos detectables respecto al resto de tipos.

Pasando a analizar los tipos de mutaciones que Pitest sí que ha podido hacer a los programas generados, podemos realizar algunas conclusiones interesantes. Fijémonos en cómo una mutación de tipo **ConstructorCallMutator** (e.d. el valor que retorna la constructora de una determinada clase) siempre es detectable en los experimentos realizados, luego un error producido en la constructora debería ser detectable muy fácilmente respecto al resto de mutaciones. Se observa también que los errores producidos en instrucciones de incremento debido a la mutación **IncrementsMutator** se detectan en la mayoría de las pruebas aplicadas. Dando un salto considerable en el DR obtenido, vemos como los tipos de mutaciones **RemoveConditionalMutator\_ORDER\_IF** y **NonvoidMethodCallMutator** ya no parecen tan fácilmente detectables.

Para finalizar el análisis, observemos un detalle a medida que incrementamos el número de parámetros por defecto. En los tipos de mutaciones **IncrementsMutator**, **RemoveConditionalMutator\_ORDER\_IF** y **NonvoidMethodCallMutator** parece que, a medida que incrementamos el valor de los parámetros, el DR obtenido va decreciendo (e.d. se van matando menos mutantes a medida que el programa “se complica”). Podría deducirse que esto ocurre, de forma general, para todo tipo de mutación, pero si observamos los tipos **MathMutator** y **MemberVariableMutator**, parece que en estos casos la detección de mutantes no está vinculada al número de parámetros que recibe el generador.

Como conclusión podemos decir que, tal y cómo se ve en las gráficas de los dos ejemplos anteriores en los que analizamos las DRs obtenidas, parece intuitivo pensar que las mutaciones que se dan en las condiciones de bifurcación *if* pueden ser relativamente más peligrosas al resto de mutaciones, ya que afectan a todo el cuerpo de la condición (incluso a otros *if* anidados que pudiese haber). Mientras que los tipos de mutaciones que se dan en las constructoras, asignaciones y, en definitiva, expresiones simples, tienden a tener una influencia relativamente menor.



## 5. Conclusiones / Conclusions

Al comenzar el proyecto nos marcamos una amplia serie de objetivos y tratamos de repartir el tiempo del que disponíamos de la mejor manera posible para cumplir el número máximo de ellos. De forma general, creemos que el trabajo se ha repartido eficientemente y que se ha dedicado el tiempo necesario para, al menos, hacer todo lo que queríamos hacer en cada una de las metas de este TFG.

Se quiso hacer una simulación en un entorno real basándonos en los tres ejemplos del artículo *Introducing Complexity to Testing* para comparar las complejidades de testing obtenidas con los resultados teóricos. Se desarrolló la aplicación que ha permitido generar árboles binarios de ejecución (representados como arrays) sobre los que introducir errores según cada una de las estrategias, aplicar pruebas y obtener sus DRs correspondientes (así como la complejidad de testing).

Pasando a la segunda fase, en términos generales podemos decir que ha sido más compleja y hemos invertido mucho más tiempo, aproximadamente los últimos cinco meses. Además, no teníamos ninguna base que nos sirviese como guía para saber por dónde debíamos ir tomando los caminos hasta hacer lo que teníamos que hacer.

Se diseñó desde cero un servicio web completo en el que hemos invertido muchas horas para que no sólo sirva para los objetivos concretos de este TFG, sino que también se use como herramienta de trabajo para todo aquel que quiera testear proyectos Java y obtener estadísticas sobre las pruebas realizadas. Es por eso que se cuidó cada detalle al desarrollarlo; desde el diseño de la base de datos hasta el código HTML y CSS para que el resultado de la interfaz web fuese lo más agradable posible.

Otro de los pilares del proyecto ha sido el programa capaz de generar otros programas Java. Éste quizá fue el mayor reto al que nos enfrentamos. Desarrollarlo ha supuesto una larga secuencia de versiones que se han ido mejorando una tras otra para que la generación se haga de la mejor manera posible y puedan obtenerse conclusiones de cierto peso sobre por qué determinadas estructuras en los programas añaden más complejidad de testing que otras.

Se quiso integrar un método que nos permitiese, tomando como base un programa Java (ya sea programado a mano o generado automáticamente), generar mutantes y poder realizar experimentos y análisis. Gracias al aprendizaje de Pitest logramos cumplir este objetivo y, obteniendo toda la información que nos proporcionaba, mostrar las gráficas que nos permitieron concluir muchos de los resultados finales de esta segunda fase.

En cuanto a las conclusiones de los experimentos realizados, las pruebas en la primera fase se realizaron satisfactoriamente y, además, podemos afirmar que los resultados obtenidos parecen ser consistentes con los teóricos (recordemos que es imposible tener una certeza total en su consistencia), lo que supone una gran satisfacción, pues vemos que los escenarios simulados se han comportado tal y cómo se esperaba que lo hiciesen. Por lo tanto, estamos conformes de que los objetivos de la primera fase del proyecto se hayan cumplido con éxito. Al mismo tiempo, creemos que los resultados que han aportado las pruebas aplicadas sobre los programas programados a mano y generados automáticamente son muy valiosos ya que, aunque esta vez no teníamos ningún punto de referencia (al menos no tan claramente definido como en la primera fase), los datos obtenidos son de una gran variedad y nos han permitido corroborar cómo algunos errores son más fácilmente detectables que otros según qué tipo de mutación haya producido dichos errores. Por último, decir que la información obtenida al aplicar pruebas sobre los programas generados de manera automática ha hecho que podamos explicar cómo determinados elementos (condiciones lógicas, expresiones aritméticas, bucles, etc.) suponen una mayor complejidad al ser testeados cuando se introducen, nuevamente gracias a las mutaciones, errores en ellos.

A pesar de las condiciones de laboratorio de nuestros experimentos, creemos que nuestras conclusiones podrían servir de base para la realización de experimentos mucho más amplios sobre programas mucho más grandes, unos experimentos que permitieran verificar empíricamente si los patrones de CT que hemos observado ocurren igualmente en grandes programas reales.

Resumiendo, quisiéramos destacar que estamos satisfechos con todos y cada uno de los objetivos. Pero no porque el resultado final haya sido satisfactorio, sino porque cada uno de los pequeños pasos que se ha dado fue verdaderamente productivo. Y al final, la suma de todos ellos ha supuesto el avance gracias al cual hemos llegado hasta aquí.

At the beginning of the project we set a wide range of objectives and tried to spread the time we had available in the best possible way to meet the maximum number of them. Overall, we believe that the work has been efficiently distributed and that time has been devoted to at least doing everything we wanted to do in each of the goals of this Bachelor's thesis.

We wanted to make a simulation in a real environment based on the three examples of the article *Introducing Complexity to Testing* to compare the complexities of testing obtained with the theoretical results. The application was developed to generate binary execution trees (represented as arrays) on which introducing errors according to each of the strategies, apply tests and obtain their corresponding DRs (as well as the complexity of testing).

Turning to the second phase, in general terms we can say that it has been more complex and we have invested much more time, approximately the last five months. Besides, we had no basis to guide us to know and where to go until we did what we had to do.

Two programs were developed by hand to perform the first tests, and seeing that we were achieving different complexities was an important milestone that prompted us to look forward to the following goals.

A complete web service was designed from scratch. We have invested many hours in it so that it will not only serve the specific objectives of this work, but will also be used as a working tool for anyone who wants to test Java projects and obtain statistics on the tests performed. That's why every detail was taken care of when developing it; from the database design to the HTML and CSS code so that the result of the web interface was as pleasant as possible.

Another of the pillars of the project has been the program capable of generating other Java programs. This was perhaps the greatest challenge we faced. Developing it has meant a long sequence of versions that have been improved one after the other so that the generation is done in the best possible way and conclusions can be drawn as to why certain structures in the programs add more testing complexity than others.

We wanted to integrate a method that would allow us, based on a Java program (either manually programmed or automatically generated), to generate mutants and be able to perform experiments and analyses. Thanks to Pitest's learning we were able to achieve this objective and, obtaining all the information he provided us, to show graphs that allowed us to conclude many of the final results of this second phase.

As for the conclusions of the experiments carried out, the tests in the first phase were carried out satisfactorily and, furthermore, we can affirm that the results obtained appear

to be consistent with the theoretical ones (let us remember that it is impossible to have total certainty in their consistency), which is a great satisfaction, since we see that the simulated scenarios have behaved as expected. Therefore, we are satisfied that the objectives of the first phase of the project have been successfully completed.

At the same time, we believe that the results provided by the tests applied to the programs programmed by hand and automatically generated are very valuable because, although this time we did not have any reference point (at least not as clearly defined as in the first phase), the data obtained are of a great variety and have allowed us to corroborate how some errors are more easily detectable than others according to which type of mutation has caused these errors. Finally, we would like to stress that the information obtained by applying tests on automatically generated programs has enabled us to explain how certain elements (logical conditions, arithmetic expressions, loops, etc.) are more complex when they are tested when they are introduced, again thanks to mutations, errors in them.

Despite the laboratory conditions of our experiments, we believe that our findings could serve as a basis for much larger experiments on much larger programmes, experiments that would allow us to verify empirically whether the patterns of TC that we have observed also occur in large real programmes.

To sum up, we would like to stress that we are satisfied with each and every one of the objectives. But not because the final result was satisfactory, but because each of the small steps taken was truly productive. And in the end, the sum of all of them has meant the progress that has brought us this far.

## 6. Reparto de tareas

### 6.1. Cristhian Rodríguez

Al igual que mis compañeros, antes de comenzar a plantearnos los objetivos del proyecto o a planificar el tiempo para el resto del curso, tuve que leer detenidamente una primera parte del artículo *A General Testability Theory* en el que hablaba de las distintas clases de complejidad de testing. Esto me sirvió para profundizar en el segundo artículo, *Introducing Complexity to Testing*, en el cual dediqué más tiempo puesto que al final del mismo se encuentran los tres ejemplos que simulamos y sobre los que realizamos análisis de complejidad de testing. Referente a esto, participé en el desarrollo sobre las ideas iniciales que fueron descartadas para llevar a cabo dichas simulaciones y, posteriormente, en el desarrollo de lo que fue la aplicación final.

Una de las primeras decisiones que tuvimos que tomar fue la de generar nuestros propios mutantes o intentar buscar una herramienta que lo hiciese por nosotros. Después de hablar con nuestro director y aclarar algunas ideas, decidimos buscar herramientas tan sólo para la segunda fase. Fue aquí donde aprendí a utilizar e integrar la herramienta Major que, como se dijo, finalmente fue sustituida por Pitest, sobre la cual aprendí cómo lleva a cabo la generación de mutantes y cómo hacer que podamos tener un cierto control sobre las mutaciones que se realizan (e.g. elegir qué tipos de mutaciones habrá en el código).

Para lograr que se cumplieran todos los objetivos marcados y que las pruebas, análisis y conclusiones se pudieran hacer en un mismo entorno, se me ocurrió la idea de desarrollar un servicio web que englobase todas las ideas que se desarrollaron. Lo que perseguía era una herramienta útil que sirviese en el futuro para todas aquellas personas que quisieran aplicar pruebas de testing a sus proyectos Java. En este aspecto, diseñé una base de datos para almacenar información acerca de los proyectos a los que se les aplicaban las pruebas para, posteriormente, mostrar los resultados. También pensé en la idea de que este entorno fuese muy visual, por lo que busqué herramientas para generar gráficas a partir de datos exportados de una base de datos. Tras buscar algunas, finalmente me descanté por Google chart, una herramienta que permite mostrar gráficas en las que analizar con claridad los resultados y que además fue fácilmente integrable con lo que ya se tenía. También tuve especial cuidado con la interfaz web; a pesar de que los conceptos que se tratan durante las dos fases pueden no ser del todo cercanos a los demás informáticos (ni siquiera para nuestros compañeros de carrera), hice especial hincapié en tratar, de alguna manera, de utilizar en la interfaz un lenguaje lo más cercano posible y ofrecer un diseño de interfaz que fuese amigable con el usuario. Esta última parte la desarrollé en los lenguajes típicos como Javascript y su librería jQuery para manejar eventos o HTML y CSS para el diseño el diseño referente al aspecto.

Además de desarrollar el servicio web y gestionar todo tipo de peticiones a él, diseñé algunos scripts en *bash* que permiten realizar un limpiado de directorios, ficheros y rutas cada vez que se terminan de realizar pruebas sobre un fichero y se desea cargar un nuevo proyecto Java. Así, el usuario que interactúe con el servicio no tiene que preocuparse de realizar ninguna tarea de gestión a este nivel, sino que él/ella simplemente se centra en subir proyectos, aplicar pruebas y analizar resultados.

Durante los últimos dos meses del TFG me encargué de realizar pruebas a distintos proyectos Java (e.d. proyectos cuyo código íbamos modificando para ver el esfuerzo que suponía testearlos). Debo decir que estas pruebas fueron muy variadas y, a medidas que las realizábamos, iban siendo cada vez más eficientes. Puesto que en esta segunda fase del proyecto no teníamos un camino claro por el que seguir para hallar qué tipo de estructura en un programa supone una mayor complejidad de testing frente a otras, las pruebas aplicadas sufrieron una evolución notable durante este periodo y fueron muchos los proyectos sobre los que realizamos experimentos (prueba de ello son los 3.633 proyectos que actualmente hay almacenados en la base de datos).

Para concluir, ayudé en la redacción de la sección **Fase II: mutaciones en Programas reales y Programas generados** de esta memoria, realizando todas las ilustraciones que se muestran, así como sus anexos y las correcciones que iba recibiendo por parte de nuestro director.

## 6.2. Yu Liu

Empecé a leer el artículo *Introducing Complexity to Testing* en el verano. Sabía que, si no entendía los conceptos de este artículo, no podría ni siquiera empezar a abordar los trabajos que debíamos hacer. Además, como el artículo está escrito en inglés y contiene muchos conceptos matemáticos con sus correspondientes demostraciones, es necesario empezar con anterioridad a leerlo (incluso antes de comenzar el curso). Aun así, no pude entenderlo hasta que tuvimos las primeras reuniones con nuestro tutor Ismael.

Después de comprender los conceptos básicos para nuestro proyecto, decidimos empezar la primera fase del TFG. En principio tuvimos dos ideas; una fue implementar una estructura de árbol binario que simula el ejemplo del árbol de ejecución, y otra fue representar el árbol de ejecución como array. Como no sabíamos cuál sería mejor, optamos por hacer las dos ideas paralelamente. O sea, por un lado, Cristhian y yo nos centrábamos en la primera idea (implementar una estructura de árbol). Por otro, Jonathan se encargó del trabajo de la segunda idea (implementar la representación de árboles en un array). A la hora de realizar la implementación, decidimos usar el lenguaje Java, puesto que hay herramientas que nos ofrecen la posibilidad de generar mutantes automáticamente. Debido a la falta de comprensión sobre algún concepto y a las dificultades de programación, nos reuníamos cada dos semanas. En cada reunión que teníamos como equipo de trabajo, no solamente

resolvíamos los problemas que teníamos a medida que avanzábamos, sino que también mostrábamos lo que cada uno había hecho, con el fin de analizar la coherencia entre nuestra práctica y la teoría del artículo *Introducing Complexity to Testing*.

Con referencia a mi parte, como veremos a continuación surgió un problema en la implementación. Lo que queríamos hacer es construir un árbol de ejecución que representase todos los posibles mutantes y optimizar la memoria y la velocidad del cálculo de DR. Así que pensábamos que, a la hora de generar mutantes, podríamos compararlos con el original y calcular el DR al mismo tiempo. De esta forma, podríamos ir borrando los mutantes que han sido comparados y contado para el resultado de DR. Sin embargo, nos dimos cuenta de que la implementación de esta idea no es tan obvia como pensamos en un principio. Principalmente porque la memoria RAM se acababa rápidamente y el tiempo de ejecución supera el tiempo límite del control de nuestro editor (Eclipse). Por lo tanto, finalmente, optamos por la segunda idea que estaba desarrollando Jonathan. Esta primera fase fue desarrollada desde mediados de octubre hasta finales de enero aproximadamente.

Empezamos la segunda fase durante el mes de enero. Después de una de las reuniones, decidí buscar algunos programas reales para estudiar los diferentes DR que pueden obtenerse entre ellos. No obstante, buscar programas reales y testearlos para obtener diferentes DR me resultó difícil. De ahí que no hubiera suficiente tiempo para realizar pruebas sobre programas que “no conocemos”, ya que para hacer pruebas sobre programas Java debemos saber qué es lo que hacen sus métodos para poder testearlo y, además, asegurarnos de que las pruebas aportarán alguna información útil (de nada nos servía aplicar pruebas a distintos programas si siempre obtenemos los mismos resultados). Por lo tanto, decidí desarrollar desde cero dos programas Javas para sacar sus DR. Como de antemano ya teníamos una intuición sobre la relación entre el valor de DR y la estructura de programa, además de desarrollar los programas también desarrollé sus tests. Durante el desarrollo utilicé Pitest, herramienta que dispone de una función que nos dice la cobertura de instrucción y de llamada de métodos. Gracias a esta herramienta, diseñé los tests de cada programa con el fin de que cada vez se llegase a una mayor cobertura. El trabajo de esta parte me llevó algo más de tiempo del que tenía previsto.

Con el fin de obtener más datos en menos tiempo, decidimos desarrollar un programa capaz de generar automáticamente otros programas Java. Antes de comenzar a escribir el código de dicho programa, me dedique unos días a analizar las distintas expresiones más comunes que puede haber en un programa (expresiones en los bucles, expresiones aritméticas o lógicas, asignaciones, bifurcaciones, etc.). Después, aproveché la gramática incontextual para estructurar el contenido del programa que queríamos generar. La dificultad más grande que tuve fue la de evitar el problema del código muerto, cuya existencia afectaría mucho nuestro estudio. Por lo tanto, decidí que el programa generado no incluyese las asignaciones de una variable a otra. Asimismo, todas las expresiones contienen un valor de

entrada (e.d. cada resultado de la expresión depende del valor de la entrada). Durante los análisis sobre los resultados de los programas generados automáticamente, se nos iban ocurriendo algunos requisitos más para mejorar la eficiencia y obtener mejores resultados. Por eso, también me encargué de las actualizaciones del Programa generador que se fueron haciendo durante los últimos dos meses del TFG. En la implementación de las nuevas versiones, también ayudé a analizar las gráficas de cada programa.

Con respecto a la redacción de la memoria, me encargué de documentar las partes referentes al desarrollo de los dos programas programados manualmente por mí y de explicar en detalle el diseño y desarrollo del Programa generador.

### 6.3. Jonathan Carrero

Como el resto de mis compañeros, el primer objetivo real fue el de comprender los dos artículos de investigación. Queríamos simular con la mayor fiabilidad posible las pruebas sobre los tres ejemplos diferentes y poder afirmar (con el mayor grado de confianza) que cada sistema comprobado de manera empírica tiende a comportarse de la misma forma que en las demostraciones teóricas. Respetando esta filosofía y simplificando las ideas lo máximo posible, diseñé el programa que se ha utilizado durante la primera fase del proyecto; un programa rápido y eficiente con el que poder realizar pruebas con el mayor alcance posible (e.d. generar un gran número de mutantes generados y de considerable profundidad). Esta primera fase nos llevó aproximadamente unos dos meses. Si bien es cierto que el desarrollo en sí (programación de código) se hizo en tres semanas, la mayor parte del tiempo se dedicó a debatir decisiones de diseño (cómo generar los mutantes, cómo almacenarlos, pensar en la forma de que el tiempo de ejecución fuese el mínimo posible, etc).

Durante la segunda fase del proyecto aprendí a utilizar Pitest y realicé, junto con mis compañeros, algunas pruebas iniciales (pues al principio, al ser un concepto totalmente nuevo para nosotros el de *mutation testing*, no sabíamos exactamente cómo funcionaba y qué teníamos que hacer para sacar el mayor partido posible a la herramienta).

Colaboré para hacer algunos retoques a la interfaz web del servicio que se estaba desarrollando, así como para realizar pruebas finales para corroborar qué tipos de mutaciones suponen una mayor complejidad al ser testeados.

Principalmente, en esta segunda fase me centré en el desarrollo y redacción de la memoria. En un principio, la memoria se quiso hacer con *LaTeX*, pero debido a que era el primer trabajo de gran envergadura al que nos enfrentamos, finalmente decidimos escoger un camino más fácil, por si finalmente no nos diese tiempo a redactar todo lo que teníamos pensado (sobre todo referente a la segunda fase). Desde el principio quise poner mucho cuidado en todo lo referente al aspecto visual de la memoria. Mientras redactaba el



documento, quería cuidar todo tipo de pequeños detalles para que el acabado final fuese lo más “perfecto” posible. Configuré el documento para que pudiésemos sacar todo el partido posible a *Word*, dividiendo el documento en secciones, haciendo los anexos necesarios, notas a pie de página, todo tipo de referencias cruzadas e incluso el interlineado, sangría y tamaño de fuente se tuvieron en cuenta (todos estos elementos, aparentemente sin importancia, al final influyen en cómo se transmite la información al lector).

También tuve especial cuidado con todas las ilustraciones y tablas que aparecen. Todos estos elementos debían perseguir el minimalismo que se ha tratado de mantener durante todo el documento; no podía haber una ilustración hecha de una determinada manera y, un poco más abajo, otra ilustración hecha con una herramienta diferente y que no se pareciese en nada a la anterior. En este sentido, aprendí a utilizar un generador de gráficas online llamado *Gliffy* que me ha permitido construir gráficas realmente útiles para completar los conceptos teóricos (a veces complicados de entender) y tratar de ser lo más sencillo y cercano posible al lector. Asimismo, para todas las capturas de código que aparecen, también se ha tratado de mantener esta misma homogeneidad.

Continuando con lo anterior, durante el último mes y medio del TFG hubo que hacer numerosas correcciones a prácticamente todos los puntos del documento. En todo momento se ha tratado de utilizar un lenguaje lo más cercano posible al lector, pero sin perder de vista los conceptos técnicos. En este sentido, nuestro director nos ayudó a subsanar esa falta de experiencia que teníamos a la hora de redactar la memoria, realizando un *feedback* constante con él que nos permitió ir mejorando poco a poco el resultado final que se presenta.

## 7. Repositorios *Github*

A lo largo de la memoria hemos ido adjuntando los enlaces donde se encuentra cada uno de los programas que hemos desarrollado. Aun así, nos gustaría dedicar este apartado para explicar brevemente cómo están organizados los repositorios que tenemos en *Github*.

Todos los repositorios utilizados durante el TFG están dentro de una organización creada por nosotros cuyo nombre es *Complexity To Testing* y que es accesible desde el siguiente enlace: <https://github.com/Complexity-To-Testing>. Dentro de esta organización encontramos los cinco repositorios en los que se encuentran la documentación y todo el código de los programas desarrollados.

- *Mutation in binary trees*: en este repositorio se encuentra la aplicación que nos ha permitido simular empíricamente los tres ejemplos del artículo y cuya explicación se vio en el apartado **Fase I: mutaciones en árboles binarios**. El código puede consultarse a través del siguiente enlace: <https://github.com/Complexity-To-Testing/Mutation-in-binary-trees>
- *Programas reales Java*: aquí se encuentran las clases Java que pertenecen al *Programa personajes* y *Programa math*, vistos en el apartado **Crear proyecto con Programa real**. El código puede consultarse a través del siguiente enlace: <https://github.com/Complexity-To-Testing/Programas-reales-JAVA>
- *Programa generador*: repositorio donde están las clases Java del *Programa generador*, gracias al cual pudimos generar automáticamente otros programas Java modificando ciertos parámetros, tal y como explicamos en el apartado **Crear proyecto con programa generado**. El código puede consultarse a través del siguiente enlace: <https://github.com/Complexity-To-Testing/Programa-generador>
- *GUI*: todo el código del servicio web que se desarrolló se encuentra aquí. El servicio web incluye todo lo visto durante segunda fase del TFG, tal y como se vio en el apartado **Fase II: mutaciones en Programas reales y Programas generados**. El código puede consultarse a través del siguiente enlace: <https://github.com/Complexity-To-Testing/GUI>
- *Documentation*: aquí se encuentra la documentación utilizada durante el TFG y la propia memoria. Los documentos pueden consultarse a través del siguiente enlace: <https://github.com/Complexity-To-Testing/Documentation>

## 8. Tecnologías utilizadas

### 8.1. Herramientas de desarrollo

#### 8.1.1. Editores

##### Atom

Atom [14] es un editor de código de fuente de código abierto para macOS, Linux, y Windows con soporte para plugins escritos en Node.js y control de versiones Git integrado, desarrollado por GitHub. Atom es una aplicación de escritorio construida utilizando tecnologías web. La mayor parte de los paquetes tienen licencias de software libre y está desarrollados y mantenidos por la comunidad de usuarios.

##### Eclipse

Eclipse es una plataforma de software compuesto por un conjunto de herramientas de programación de código abierto multiplataforma para desarrollar lo que el proyecto llama "Aplicaciones de Cliente Enriquecido", opuesto a las aplicaciones "Cliente-liviano" basadas en navegadores. Esta plataforma típicamente ha sido usada para desarrollar entornos de desarrollo integrados (del inglés IDE), como el IDE de Java llamado Java Development Toolkit (JDT) y el compilador (ECJ) que se entrega como parte de Eclipse (y que son usados también para desarrollar el mismo Eclipse).

##### IntelliJ IDEA

IntelliJ IDEA [15] es un entorno de desarrollo integrado (IDE) para el desarrollo de programas informáticos. Es desarrollado por JetBrains (anteriormente conocido como IntelliJ), y está disponible en dos ediciones: edición para la comunidad y edición comercial. IntelliJ IDEA no está basada en Eclipse como MyEclipse u Oracle Enterprise Pack para Eclipse.

##### SublimeText

SublimeText [16] es un editor de texto y editor de código fuente escrito en C++ y Python para los *plugins*. Desarrollado originalmente como una extensión de Vim, con el tiempo fue creando una identidad propia, por esto aún conserva un modo de edición tipo vi llamado Vintage mode.

#### 8.1.2. Software

##### Gliffy

Gliffy [17] es un software para hacer diagramas a través de una aplicación HTML5 basada en la nube. Los tipos de diagramas disponibles son diagramas UML, planos de planta, diagramas de Venn, diagramas de flujo y otros tipos de diagramas en línea. Los diagramas Gliffy pueden ser compartidos y editados por los usuarios en tiempo real.

## Sed

Sed [18] es una potente herramienta que analiza y transforma el texto, usando un lenguaje de programación simple y compacto. Fue desarrollado de 1973 a 1974 por Lee E. McMahon de Bell Labs, y está disponible hoy para la mayoría de los sistemas operativos.

## XAMPP

XAMPP es un paquete de software libre que consiste principalmente en el sistema de gestión de bases de datos MySQL, el servidor web Apache y los intérpretes para lenguajes de script PHP y Perl. El nombre es en realidad un acrónimo: **X** (para cualquiera de los diferentes sistemas operativos), **A**pache, **M**ariaDB/MySQL, **P**HP, **P**erl. A partir de la versión 5.6.15, XAMPP cambió la base de datos MySQL por MariaDB, un fork de MySQL con licencia GPL. XAMPP trae incorporado Apache, MariaDB y PhpMyAdmin, que explicaremos a continuación.

### Apache

Apache es un servidor web HTTP de código abierto para plataformas Unix (BSD, GNU/Linux, etc.), Microsoft Windows, Macintosh y otras, que implementa el protocolo HTTP/1.1 y la noción de sitio virtual.

### MariaDB

MariaDB es un sistema de gestión de bases de datos derivado de MySQL con licencia GPL (General Public License). Introduce dos motores de almacenamiento nuevos, uno llamado Aria —que reemplaza con ventajas a MyISAM— y otro llamado XtraDB —en sustitución de InnoDB—. Tiene una alta compatibilidad con MySQL ya que posee las mismas órdenes, interfaces, APIs y bibliotecas, siendo su objetivo poder cambiar un servidor por otro directamente.

### PhpMyAdmin

PhpMyAdmin [19] es una herramienta escrita en PHP con la intención de manejar la administración de MySQL a través de páginas web, utilizando Internet. Actualmente puede crear y eliminar Bases de Datos, crear, eliminar y alterar tablas, borrar, editar y añadir campos, ejecutar cualquier sentencia SQL, administrar claves en campos, administrar privilegios, exportar datos en varios formatos y está disponible en 72 idiomas.

### **Node.js**

Node.js [20] es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor (pero no limitándose a ello) basado en el lenguaje de programación ECMAScript, asíncrono, con I/O de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.

### **Maven**

Maven [21] es una herramienta de software para la gestión y construcción de proyectos Java creada por Jason van Zyl, de Sonatype, en 2002. Es similar en funcionalidad a Apache Ant (y en menor medida a PEAR de PHP y CPAN de Perl), pero tiene un modelo de configuración de construcción más simple, basado en un formato XML.

### **JUnit**

JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera. Es decir, en función de algún valor de entrada se evalúa el valor de retorno esperado; si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba; en caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente.

### **Pitest**

Pitest es un sistema de pruebas de mutación de última generación que proporciona cobertura de pruebas para Java y la JVM. Es rápido, escalable y se integra con herramientas modernas de prueba y construcción.

### **Office**

Office [22] es una suite ofimática que abarca el mercado completo en Internet e interrelaciona aplicaciones de escritorio, servidores y servicios para los sistemas operativos Microsoft Windows, Mac OS X, iOS y Android.

## **8.2. Herramientas de gestión**

### **8.2.1. Control de versiones**

#### **Git**

Git [23] es un software de control de versiones diseñado por Linus Torvalds, pensando en la eficiencia y la confiabilidad del mantenimiento de versiones de aplicaciones cuando éstas tienen un gran número de archivos de código fuente. Su propósito es llevar registro de los cambios en archivos de computadora y coordinar el trabajo que varias personas realizan sobre archivos compartidos.

### **Github**

Github es una forja (plataforma de desarrollo colaborativo) para alojar proyectos utilizando el sistema de control de versiones Git. Se utiliza principalmente para la creación de código fuente de programas de computadora.

### **SourceTree**

SourceTree [24] es un cliente gratuito de Mercurial y Git para Windows y Mac que ofrece una interfaz gráfica para tus repositorios de Hg y Git.

## **8.2.2. Organización del proyecto**

### **Drive**

Drive [25] es un servicio de alojamiento de archivos que fue introducido por la empresa estadounidense Google el 24 de abril de 2012. Es el reemplazo de Google Docs que ha cambiado su dirección URL, entre otras cualidades. Cada usuario cuenta con 15 gigabytes de espacio gratuito para almacenar sus archivos, ampliables mediante diferentes planes de pago.

### **Meet**

Meet [26] es una aplicación de Google que permite realizar videoconferencias. Surge como reemplazo a Hangouts en el terreno profesional y está disponible tanto en versión web como para dispositivos móviles.

## Anexo I: Hipótesis en las pruebas

- *Independency hypothesis (IH)*: por cada dos aristas X, Y del árbol de ejecución de la IUT, se asume que el evento que hace que X produzca una salida defectuosa es independiente del evento que hace que Y produzca una salida defectuosa. Hay que tener en cuenta que esto no es cierto en general, ya que dos aristas distintas X e Y del árbol de ejecución podrían representar la ejecución de la misma estructura lógica de la IUT (e.g. ejecutar la misma línea de código, llamar a la misma función, etc.).
- *Almost correct hypothesis (ACH)*: la IUT es prácticamente correcta, por lo que su árbol de ejecución sólo puede contener una arista donde la salida es incorrecta. De nuevo, esta suposición no es cierta en general, ya que incluso una sola estructura lógica incorrecta puede ser la fuente de fallos en varias situaciones diferentes (e.d. varias aristas del árbol de ejecución).
- *Mostly initially correct hypothesis (MICH)*: el desarrollador de la IUT (e.g. un programador) ya aplicó varios casos de prueba aleatorios a la IUT. Esto significa que las fallas a pequeñas profundidades en el árbol de ejecución son menos probables que los errores a una mayor profundidad, aunque a medida que se va profundizando, la probabilidad de los fallos aumenta suavemente en lugar de abruptamente. En nuestro modelo, asumiremos que cada error reduce el peso de la función correspondiente, y los fallos más cercanos a la raíz del árbol de ejecución lo reducen aún más.
- *Initially correct hypothesis (ICH)*: el desarrollador de la IUT ya aplicó varios casos de prueba, aunque esta vez se realizaron de forma más o menos sistemática hasta alcanzar una profundidad fija desconocida. Por lo tanto, esperamos que los fallos de la IUT se extiendan abruptamente más allá de alguna profundidad desconocida. En nuestro modelo, asumiremos que el peso de cada función se ve afectado por el primer nivel del árbol de ejecución donde hay un error.
- *The deeper the better hypothesis (DBH)*: la IUT es prácticamente correcta con un máximo de un error y asumimos que la probabilidad de observar dicho error disminuye a medida que se profundiza en el árbol de ejecución. La razón de esta suposición es que, después de realizar una ejecución larga sin observar ningún error, la probabilidad de observar un error que no haya sido descubierto anteriormente es menor, ya que se reduce la cantidad de fuentes de errores potenciales que aún no han sido comprobadas. En nuestro modelo, esto se representará asignando un peso menor cuanto más profundo se encuentre el error.

## Anexo II: Tipos de mutaciones en Pitest

Dentro del abanico de posibilidades que tiene Pitest a la hora de realizar mutaciones en el código fuente de un programa, algunas vienen activadas por defecto, y otras, por ser más propensas a generar fallos críticos, desactivadas. Consideramos que no tiene cabida

poner todos los tipos de mutaciones, por lo que adjuntamos una tabla con los nombres y enlaces a la página web de Pitest, para que puedan ser consultados.

Nombre / Enlace	Se activa mediante	Por defecto
<a href="#">Conditionals Boundary Mutator</a>	CONDITIONALS_BOUNDARY	✓
<a href="#">Increments Mutator</a>	INCREMENTS	✓
<a href="#">Invert Negatives Mutator</a>	INVERT_NEGS	✓
<a href="#">Math Mutator</a>	MATH	✓
<a href="#">Negate Conditionals Mutator</a>	NEGATE_CONDITIONALS	✓
<a href="#">Return Values Mutator</a>	RETURN_VALS	✓
<a href="#">Void Method Call Mutator</a>	VOID_METHOD_CALLS	✓
<a href="#">Constructor Call Mutator</a>	CONSTRUCTOR_CALLS	✗
<a href="#">Inline Constant Mutator</a>	INLINE_CONSTS	✗
<a href="#">Non Void Method Call Mutator</a>	NON_VOID_METHOD_CALLS	✗
<a href="#">Remove Conditionals Mutator</a>	REMOVE_CONDITIONALS	✗
<a href="#">Experimental Member Variable Mutator</a>	EXPERIMENTAL_MEMBER_VARIABLE	✗
<a href="#">Experimental Switch Mutator</a>	EXPERIMENTAL_SWITCH	✗

Tabla 2: Tipos de mutaciones en Pitest

### Anexo III: Características de Java

Ya que Java es un lenguaje muy conocido y extendido en el mundo informático, consideramos que no tiene cabida poner todas sus características. Si se desea profundizar más sobre ellas, pueden consultarse en el siguiente [enlace](#). Lo que sí nos interesa es nombrar las características que, particularmente, a nosotros nos ha hecho decantarnos por este lenguaje de programación, sin olvidar, por supuesto, que ha sido uno de los lenguajes más estudiados en nuestras asignaturas de programación.

- Lenguaje totalmente orientado a objetos.
- Disponibilidad de un amplio conjunto de bibliotecas.
- Interpretado y compilado a la vez.
- Indiferente a la arquitectura.

### Anexo IV: Métodos principales usados en la estructura de *ArrayList*

Método	Descripción
<code>add(E e)</code>	Appends the specified element to the end of this list.
<code>add(int index, E element)</code>	Inserts the specified element at the specified position in this list.
<code>contains(Object o)</code>	Returns true if this list contains the specified element.
<code>get(int index)</code>	Returns the element at the specified position in this list
<code>isEmpty()</code>	Returns <i>true</i> if this list contains no elements.
<code>remove(int index)</code>	Removes the element at the specified position in this list.



<code>set(int index, E element)</code>	Replaces the element at the specified position in this list with the specified element.
<code>size()</code>	Returns the number of elements in this list.

Tabla 3: Método principales de *ArrayList*

## Anexo V: Efecto frontera

El *efecto frontera* se produce debido a que resulta imposible generar un número arbitrariamente alto de árboles y niveles de profundidad. Ya que nuestros medios son limitados, tan sólo podemos realizar experimentos con un pequeño número de muestras (que, como se dijo, fue de 100.000) y entre 6 y 7 niveles de profundidad.

Al estar tratando de representar un conjunto infinito de mutantes con un conjunto finito de ellos, el resultado es que, en nuestros experimentos, es posible llegar a DR 1, lo que sería imposible sobre el conjunto infinito. De hecho, con un test suite completo (e.d. que explore todas las ramas hasta un nivel de profundidad igual a la profundidad considerada en los mutantes generados) siempre alcanzaremos un  $DR = 1$ .

## Anexo VI: Estructura de carpetas

Vamos a explicar la estructura de directorios que tiene el proyecto. El proyecto se encuentra alojado en Git y se puede acceder a él a través del siguiente [enlace](#).

🚦 *GUI-Master*: directorio que contiene todo el proyecto. Dentro se encuentra el resto de los directorios que veremos a continuación. Además, también almacena los scripts *.sh* que se encargan de controlar las rutas de los ficheros, modificaciones de cadenas de caracteres y limpieza de proyectos anteriormente cargados y/o ejecutados, así como los ficheros *.js* encargados de poner en marcha el proyecto y almacenar la configuración de la base de datos.

- *controllers*: almacena los ficheros *.js* que se encargan de llevar a cabo la gestión de peticiones (ya sean tipo GET o POST) que realiza el usuario a través de la interfaz web. Si el usuario introduce parámetros, estos ficheros contienen funciones que los recogen y los tratan como sea necesario. Además, también realizan peticiones a la capa de integración, es decir, son los encargados de llamar a los DAO<sup>14</sup> para obtener datos de la base de datos y subirlos de nuevo a las capas superiores para que finalmente sean mostradas al usuario.

---

<sup>14</sup> En software de computadores, un objeto de acceso a datos (en inglés, Data Access Object, abreviado DAO) es un componente de software que suministra una interfaz común entre la aplicación y uno o más dispositivos de almacenamiento de datos, tales como una base de datos o un archivo. El término se aplica frecuentemente al Patrón de diseño Object.

- *generadorMutantes.JAVA*: contiene el proyecto Java que, a su vez, contiene el código del programa que se ha generado de manera automática.
- *generadorProgramas.JAVA*: contiene el proyecto Java que, a su vez, contiene el código de la aplicación que permite generar programas Java de manera automática y generar tests para aplicarlos en los programas generados.
- *helpers*: contiene algunas funciones programadas en Javascript que permiten facilitar nuestra interacción con la base de datos a la hora de realizar las queries e importar/exportar datos.
- *integration*: contiene las funciones que acceden a la base de datos para importar/exportar datos. Estas funciones son llamadas por los controladores que, tal y como hemos visto anteriormente, se encuentran en la carpeta controllers.
- *node\_modules*: diferentes módulos de node.js que hemos incorporado para un desarrollo más eficiente.
- *public*: aquí se encuentran todos los ficheros que forman la interfaz web y lo que el usuario ve cuando accede a la página web.

## Anexo VII: Cobertura de código

Como hemos destacado con anterioridad, el esfuerzo de diseñar tests de tal forma que nos aseguremos que cada vez se alcanza un DR mayor (y que además ese DR es considerablemente cercano a 1) puede llegar a ser muy alto. Durante el diseño de los tests hemos tratado de diseñarlos lo mejor posible para que, al menos, su cobertura de código sea cercana al 100% (lo que, recordemos, no implica alcanzar un  $DR = 1$ ). A continuación, vamos a ver cuatro ilustraciones que reflejan la cobertura de código (proporcionada por Pitest) alcanzada en cada paquete del proyecto Java de cada uno de los dos programas reales (recordemos, *Programa personaje* y *Programa math*). Cada uno de los cuatro paquetes contienen las diez clases Java que testean los programas: un paquete para el *Programa math* y tres paquetes para el *Programa personaje*.

- Tests aplicados al paquete math: contiene todos los tests que se aplican sobre el paquete *math*.







Coverage				
Counter	Coverage	Covered	Missed	Total
Instructions	 100.0 %	725	0	725
Branches	 89.7 %	52	6	58
Lines	 100.0 %	116	0	116
Methods	 100.0 %	17	0	17
Types	 100.0 %	3	0	3
Complexity	 87.8 %	43	6	49

Ilustración 39: Cobertura de código en paquete *Math*

- Tests aplicados al paquete figura2D: contiene todos los tests que se aplican sobre el paquete *figura2D*.







Coverage				
Counter	Coverage	Covered	Missed	Total
Instructions	 100.0 %	416	0	416
Branches	 81.2 %	13	3	16
Lines	 100.0 %	92	0	92
Methods	 100.0 %	31	0	31
Types	 100.0 %	4	0	4
Complexity	 92.3 %	36	3	39

Ilustración 40: Cobertura de código en paquete *figura2D*

- Tests aplicados al paquete figura3D: contiene todos los tests que se aplican sobre el paquete *figura3D*.







Coverage				
Counter	Coverage	Covered	Missed	Total
Instructions	 98.7 %	1,193	16	1,209
Branches	 100.0 %	90	0	90
Lines	 96.4 %	215	8	223
Methods	 91.1 %	41	4	45
Types	 100.0 %	5	0	5
Complexity	 95.6 %	86	4	90

Ilustración 41: Cobertura de código en paquete *figura3D*

- Tests aplicados al paquete personaje: contiene todos los tests que se aplican sobre el paquete *personaje*.







Coverage				
Counter	Coverage	Covered	Missed	Total
Instructions	 100.0 %	368	0	368
Branches	 100.0 %	14	0	14
Lines	 100.0 %	75	0	75
Methods	 100.0 %	31	0	31
Types	 100.0 %	4	0	4
Complexity	 100.0 %	38	0	38

Ilustración 42: Cobertura de código en paquete *personaje*

## Anexo VIII: Parámetros del *Programa generador*

Pasamos a describir cada uno de los parámetros que pueden configurarse en el Programa generador de programas que hemos desarrollado. Parámetros que, recordemos, son introducidos a través de la interfaz web mediante un formulario HTML.

- **anidaciónIf**: número entero que indica el nivel de anidación en una condición *if*. Para un valor de 1, obtenemos un solo *if*. Para un valor 2, obtenemos dos *if*, uno dentro de otro. Y así para cualquier  $n$ .
- **anidacionFor**: análogo al anterior, pero con bucles *for*.
- **anidacionWhile**: análogo al anterior, pero con bucles *while*.
- **sizeFor**: determina el número de iteraciones que realiza un bucle *for*.
- **sizeWhile**: análogo al anterior, pero con bucles *while*.
- **size\_expLogics**: especifica el número de operadores lógicos en una expresión lógica.
- **size\_expArit**: especifica el número de operadores aritméticos en una expresión aritmética.
- **size\_cond**: especifica el número de operadores lógicos que aparecen en las condiciones de las expresiones *while* e *if*.
- **num\_function**: especifica el número de funciones que se va a generar.
- **decisión\_inputs**: es una lista de números enteros. Cada uno de ellos indica el método concreto del programa que se va a ejecutar (en base al parámetro anteriormente visto).
- **ini**: especifica el inicio de la comprobación entre el resultado de un programa original y el resultado de los mutantes (porque el resultado es almacenado como una lista de elementos).
- **fin**: análogo al anterior, pero para el final de la comprobación.
- **ifsAniCuerpoBucle**: indica la anidación de *if* que habrá dentro de los cuerpos de los bucles.

- **num\_exp\_seguida:** establece una serie de expresiones consecutivas. Todas ellas son asignaciones a variables (simples o en las que haya que realizar alguna operación matemática, como por ejemplo multiplicar dos números).
- **num\_tests:** indica el número de tests que se generan. Los tests generados tienen un número asociado y, a medida que se incrementa el valor asociado, la cobertura de código aumenta.
- **aleatorio:** un *check* que indica si se desea que la estructura (e.d. los parámetros) se establezcan de forma aleatoria. Podemos diferenciar dos posibilidades:
  - a) Si el *check* está marcado (e.d. si su valor es 1), entonces el generador toma los valores de **size\_for**, **size\_while**, **size\_expArit**, **size\_expLogics**, **num\_exp\_seguida**, **size\_cond**, **anidacionIf**, **anidacionFor** y **anidacionWhile** como el número máximo que puede generar para cada parámetro correspondiente (el número mínimo debe ser 1 para cualquiera de ellos). En otras palabras: el generador toma un valor aleatorio en el intervalo **[1, atributo]**.
  - b) Si el *check* no está marcado (e.d. su valor es 0), entonces el generador va a tomar como valor de los atributos los valores que se hayan introducido en el formulario.

## Anexo IX: Gramática Incontextual

En informática, una gramática libre de contexto [27] es una gramática formal (e.d. una estructura matemática con un conjunto de reglas de formación que definen las cadenas de caracteres admisibles en un determinado lenguaje formal) en la que cada regla de producción es de la forma:  $V \rightarrow w$ , donde  $V$  es un símbolo no terminal y  $w$  es una cadena de terminales y/o no terminales. El término libre de contexto se refiere al hecho de que el no terminal  $V$  puede siempre ser sustituido por  $w$  sin tener en cuenta el contexto en el que ocurra. Un lenguaje formal es libre de contexto si hay una gramática libre de contexto que lo genera. Veamos un ejemplo para entenderlo mejor. Se presenta una gramática libre de contexto para expresiones entera algebraicas sintácticamente correctas sobre las variables  $x$ ,  $y$  y  $z$ .

$$S \rightarrow x \mid y \mid z \mid S + S \mid S - S \mid S * S \mid S / S \mid (S)$$

Esta gramática generaría, por ejemplo, la cadena:  $(x + y) * x - z * y / (x + x)$

## Anexo X: Secuencia de tareas del fichero *preprocesar.sh*

Vamos a ver en profundidad cómo el script *preprocesar.sh* se encarga de realizar tres tareas muy importantes para que no haya conflictos entre los distintos proyectos que se van cargando a través del servicio web.

En primer lugar (correspondiente al paso 3 que veíamos durante la explicación de la sección de Preprocesado), se hace un borrado de las rutas de las clases Java del proyecto anteriormente cargado para que, cuando el usuario elija cargar un nuevo proyecto (ya sea un programa real o uno generado), se vuelva a configurar el fichero *pom.xml*<sup>15</sup> con las rutas de las clases del nuevo proyecto.

```
#####
#
# 3. Preparamos el fichero de configuración      #
# con los mutantes que se van a generar          #
#
#####

for idMutante in $MUTANTES_GENERAR
do
    echo $idMutante
    echo "${LISTA_MUTANTES[${idMutante}]}"
    echo "<mutator>${LISTA_MUTANTES[${idMutante}]}</mutator>" >> $FILE_POM
done
echo "</mutators>"
    <failWhenNoMutations>false</failWhenNoMutations>
    <outputFormats>
        <param>XML</param>
        <param>HTML</param>
        <param>CSV</param>
    </outputFormats>
    <targetClasses>" >> $FILE_POM
```

A continuación, se procede a descomprimir el fichero *Classes.zip* (los cuales contienen las clases Java que formarán el nuevo proyecto) y se añaden al nuevo proyecto PIT.

```
#####
#
# 4. Preparamos el fichero de configuración con las clases del programa #
#
#####

# 4.1. Comprobamos que el fichero zip Classes.zip es un fichero regular
if [ -f $FILE_CLASSES_ZIP ]; then
    echo "File $FILE_CLASSES_ZIP exists."

    # 4.2. Descomprimos el fichero Classes.zip en el directorio
    # donde estan clases del proyecto PIT
    unzip -o $FILE_CLASSES_ZIP -d $DIR_PROYECTOJAVA_CLASSES

    # 4.3. Para cada fichero clase, agregamos de manera recursiva
    # la ruta de cada fichero en el fichero de configuración pom.xml
    cd $DIR_PROYECTOJAVA_CLASSES
    procAddFilesClassesToFilePomRec "."
    cd -

    # 4.4. Agregamos las etiquetas de cierre de la classes
    # y tambien agregamos las etiquetas de apertura para
    # los tests que se agregaran en el punto 3
    echo "</targetClasses><targetTests>" >> $FILE_POM
else
    echo "File $FILE_CLASSES_ZIP does not exist."
fi
```

<sup>15</sup> Dentro de un proyecto Maven, el fichero *pom.xml* –Project Object Model– es su “unidad” principal y contiene toda la información acerca del proyecto, fuentes utilizadas, tests, dependencias, plugins, versión con la que se trabaja, etc. Antes de que ejecutemos una tarea o proyecto, Maven *mira* su fichero *pom.xml* correspondiente para conocer toda esta información.

Por último, se hace exactamente lo mismo que en el paso anterior sólo que esta vez se descomprime el fichero *Tests.zip* (que contiene los tests, programados en Java, que se utilizarán para realizar pruebas sobre el propio proyecto).

```
#####
#
# 5. Preparamos el fichero de configuración con los test del programa #
#
#####

# 5.1. Comprobamos que el fichero Test.zip existe
if [ -f $FILE_TESTS_ZIP ]; then
    echo "File $FILE_TESTS_ZIP exists."

    # 5.2. Descomprimos el fichero Tests.zip en el directorio
    # de donde estan los tests del proyecto PIT
    unzip -o $FILE_TESTS_ZIP -d $DIR_PROYECTOJAVA_TEST

    # 5.3. Para cada fichero test, agregamos de manera recursiva
    # la ruta de cada fichero en el fichero de configuración pom.xml
    cd $DIR_PROYECTOJAVA_TEST
    procAddFilesTestToFilePomRec "."
    cd -
    # 5.4. Terminamos de construir el fichero de configuración
    # con las etiquetas de cierre correspondientes
    echo "</targetTests></configuration></plugin></plugins></build></project>" >> $FILE_POM
else
    echo "File $FILE_TESTS_ZIP does not exist."
fi
```

## Anexo XI: Comandos Unix

Nombre	Descripción	Opciones
if	Permite ejecutar una secuencia de comandos dependiendo de la condición especificada.	<ul style="list-style-type: none"> <li>• -d <i>fichero</i>: es un directorio.</li> <li>• -e <i>fichero</i>: existe el fichero.</li> <li>• -f <i>fichero</i>: es un fichero regular.</li> </ul>
rm	Permite borrar ficheros o directorios.	<ul style="list-style-type: none"> <li>• -r <i>directorio</i>: hace un borrado recursivo.</li> </ul>
mkdir	Permite crear directorios.	
cp	Permite copiar un fichero.	
unzip	Permite descomprimir ficheros .zip.	<ul style="list-style-type: none"> <li>• -o <i>fichero</i>: sobrescribe el fichero en el directorio destino.</li> <li>• -d <i>fichero</i>: descomprime el fichero en un directorio pasado por parámetro.</li> </ul>
Argumentos	Los argumentos que se le pasan a una función o script son almacenados en variables reservadas que aumentan secuencialmente y son llamadas con \$. Por ejemplo: \$1, \$2, \${10}, \${21}, etc.	



<b>cut</b>	Permite cortar cadenas de caracteres o campos con la posibilidad de usar delimitadores.	<ul style="list-style-type: none"> <li>• <b>-f <i>campos</i></b>: selecciona únicamente los campos especificados.</li> <li>• <b>-d <i>delimitadores</i></b>: utiliza los delimitadores especificados.</li> </ul>
<b>&gt;&gt;</b>	Redirige la salida estándar al final del fichero pasado como parámetro.	
<b>&gt;</b>	Redirige la salida estándar al principio del fichero pasado como parámetro, machacando lo que hubiese anteriormente.	
<b>rev</b>	Invierte el texto pasado como parámetro.	

Tabla 4: Comando Unix utilizados

## Anexo XII: Características de la base de datos

Servidor de base de datos:

- Servidor local: Localhost vía UNIX socket
- Tipo de servidor: MariaDB
- Versión de servidor: 10.1.30–MariaDB–Source distribution
- Versión del protocolo: 10
- Usuario: root@localhost
- Conjunto de caracteres de servidor: UTF–8 Unicode (utf8)

Servidor web

- Apache 2.4.29 UNIX
- OpenSSL 1.0.2n
- PHP 7.0.27
- Perl 5.1.3

Información almacenada en las tablas:

- *proyectos*: es una tabla que se utiliza para tener identificado cada proyecto Java que se importa en el servicio web. En dicha tabla lo único que se almacena es el nombre del proyecto y un identificador asociado.
- *test\_proyecto*: se almacena un identificador para cada test y el identificador del proyecto al que dicho test es aplicado. Además, también se guarda el nombre del test, así como el número de mutantes totales, número de mutantes matados y DR obtenido. Por último, también se almacena el tiempo que ha tardado en aplicarse sobre el proyecto.
- *mutante\_test\_proyecto*: esta tabla es utilizada para almacenar información sobre los mutantes generados. Cada mutante se identifica de manera única y



también se guarda el identificador del proyecto Java al cual pertenece, la clase concreta en la que se encuentra y el identificador del test que se le ha aplicado. Además, se guarda el tipo de mutación que es (según los tipos de mutaciones que vimos en Pitest) y su estado tras haber aplicado el test asociado.

## 9. Referencias

- [1] Ismael Rodríguez, Fernando Rosa-Velardo and Fernando Rubio, “Introducing Complexity to Testing,” *En proceso de revisión*.
- [2] Ismael Rodríguez, Luis Llana and Pablo Rabanal, “A General Testability Theory”, *IEEE Transactions on software engineering*, Vol. 40, no. 9, 862-894. 2014.
- [3] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34-41. April 1978.
- [4] A. Jefferson Offutt. Investigations of the software testing coupling effect. *ACM Trans. Softw. Eng. Methodol.* 1, 5-20. 1992.
- [5] Software Pitest. Disponible: <http://pitest.org>
- [6] Software Eclipse. Disponible: <https://www.eclipse.org>
- [7] Software Muclipse. Disponible: <http://muclipse.sourceforge.net>
- [8] Software Major. Disponible: <http://mutation-testing.org>
- [9] Software JUnit. Disponible: <https://junit.org/junit5>
- [10] Software Apache. Disponible: <https://www.apache.org>
- [11] Software XAMPP. Disponible: <https://www.apachefriends.org/es/index.html>
- [12] Software MariaDB. Disponible: <https://mariadb.org>
- [13] Software Github. Disponible: <https://github.com>
- [14] Software Atom. Disponible: <https://atom.io>
- [15] Software IntelliJ IDEA. Disponible: <https://www.jetbrains.com/idea>
- [16] Software SublimeText. Disponible: <https://www.sublimetext.com>
- [17] Software Gliffy. Disponible: <https://www.gliffy.com>
- [18] Software Sed. Disponible: <https://www.gnu.org/software/sed>
- [19] Software PhpMyAdmin. Disponible: <https://www.phpmyadmin.net>
- [20] Software Node.js. Disponible: <https://nodejs.org/es>
- [21] Software Maven. Disponible: <https://maven.apache.org>
- [22] Software Office. Disponible: <https://www.office.com>
- [23] Software Git. Disponible: <https://git-scm.com>
- [24] Software SourceTree. Disponible: <https://www.sourcetreeapp.com>
- [25] Software Drive. Disponible: [https://www.google.com/intl/es\\_ALL/drive](https://www.google.com/intl/es_ALL/drive)
- [26] Software Meet. Disponible: <https://meet.google.com>
- [27] Paul Ammann, Jeff Offutt, “Introduction to Software Testing”. Cambridge University Press. 2013.

